A Scalable Architecture for Reprioritizing Ordered Parallelism

Gilead Posluns, Yan Zhu, Guowei Zhang, Mark C. Jeffrey

ISCA 2022

UNIVERSITY OF TORONTO

HUAWEI

Ordered algorithms use priority schedules

```
pq = init();
while (!pq.empty())
      task, ts = pq.dequeueMin()
      task(ts)
```

Priority schedules accelerate convergence

Dijkstra's SSSP

Breadth First Search

Residual Belief Propagation

Priority schedules are correct

Minimum Spanning Forest

KCore

Set Cover

Maximal Independent Set

Priority schedules are powerful, but hard to parallelize

Hive parallelizes priority updates

Hive builds on Swarm to provide a parallel **priority update** operation in speculative task-parallel hardware

Hive speculates eagerly on data, control, and **scheduler dependences**

Hive achieves >100x speedup over parallel software, and up to 2.8x over prior speculative hardware at 256 cores

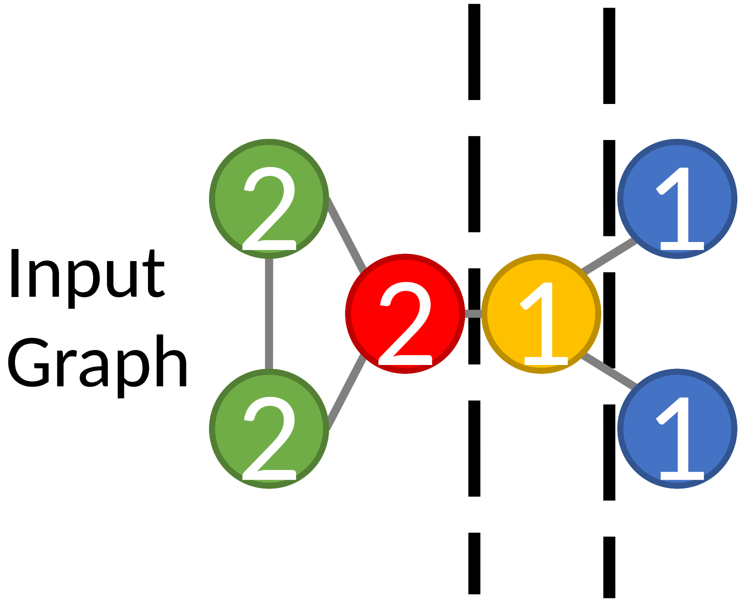
Understanding Priority Updates

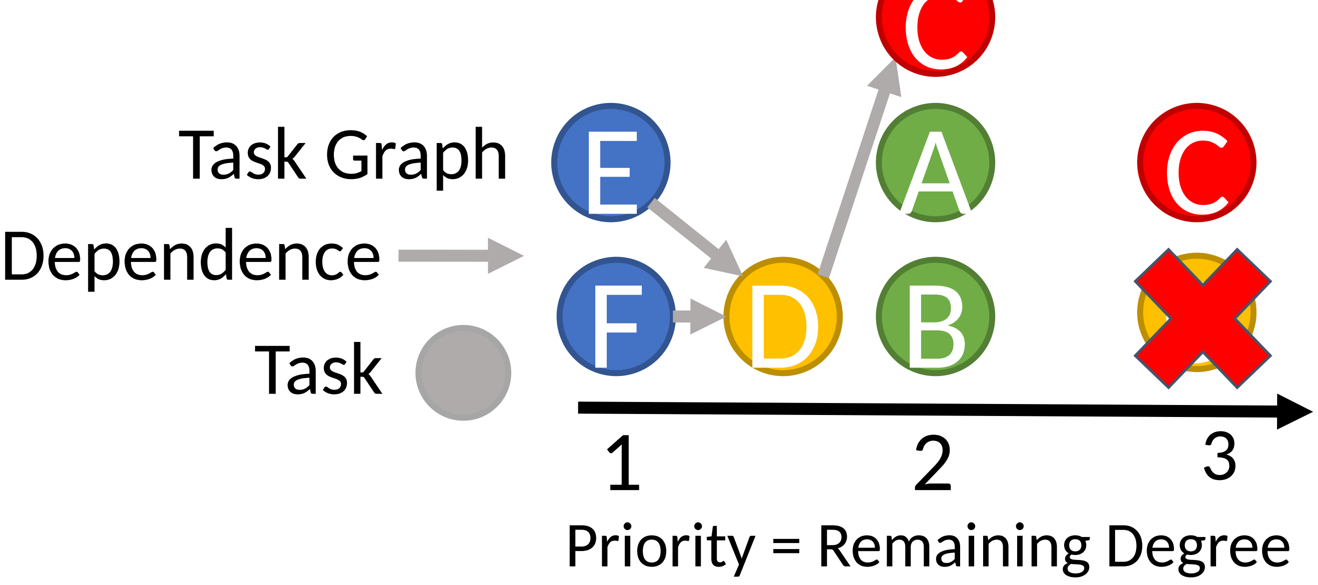
KCore requires priority updates

Max core of a vertex ≈ “importance” [Malliaros et al. VLDB 29]

To find: repeatedly remove lowest degree vertex

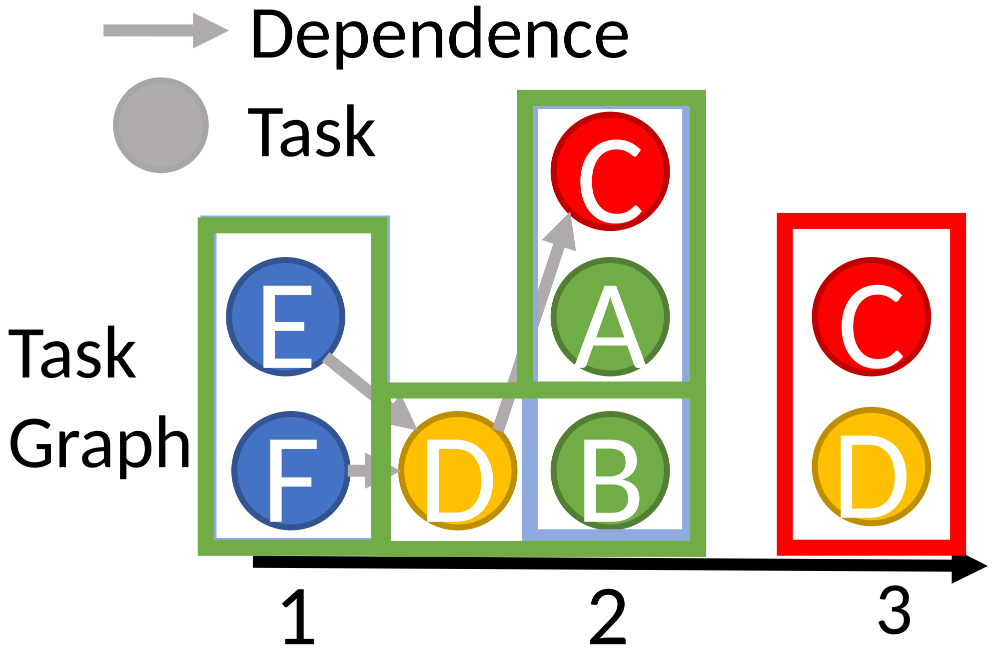
```
PriorityQueue pq;
for (int v: G.V)
  pq.enqueue(v, G.degree[v])
while (!pq.empty()) {
  int v, int prio = pq.dequeueMin();
  coreness[v] = prio;
  for (int nbr : G.edges[v])
    pq.decrementPrio(nbr)
}}
```





Where’s the parallelism in KCore?

- Bulk-Synchronous [Dhulipala et al. SPAA‘17] [Dadu et al. ISCA’21]
  - Effective when many tasks per barrier
  - Nearly sequential when few tasks per barrier
- Relaxed [Khan et al HPCA’22] [Yesil et al. SC’19] [Dadu et al. ISCA’21]
  - Can always find parallelism
  - loses efficiency as it scales
  - Not always correct
- Speculation [Blelloch et al. PPoPP’12][Jeffrey et al. MICRO’15]
  - Always finds parallelism
  - Maintains strict ordering
  - SW speculation has high overheads



Our goal is to support priority updates in speculative parallel hardware

Swarm [Jeffrey et al. MICRO'15] speculates without updates

Task-Based Execution Model

- Programs consist of timestamp-ordered tasks
- Tasks appear to execute in timestamp order
- Scheduler is **only** accessed with enqueues

```
while (!pq.empty())
      task, ts = pq.dequeueMin()
      task(ts)
```

```
swarm::enqueue(
      fn,   //what to do
      ts,   //when to do it
      args //what to do it with)
```

Swarm's execution model does not support priority updates

Swarm KCore is inefficient (i.e., without updates)

```
PriorityQueue pq;
i
for (int v: G.V) {
  p                              ;
  pq.enqueue(v, prios[v]);
}
while (!pq.empty()) {
  int v, int prio = pq.dequeueMin();
  i                             e;
  coreness[v] = prio;
  for (int nbr : G.edges[v])
    if (prios[nbr] > prio) {
      |
      pq.enqueue(nbr, prios[nbr])
}}
```

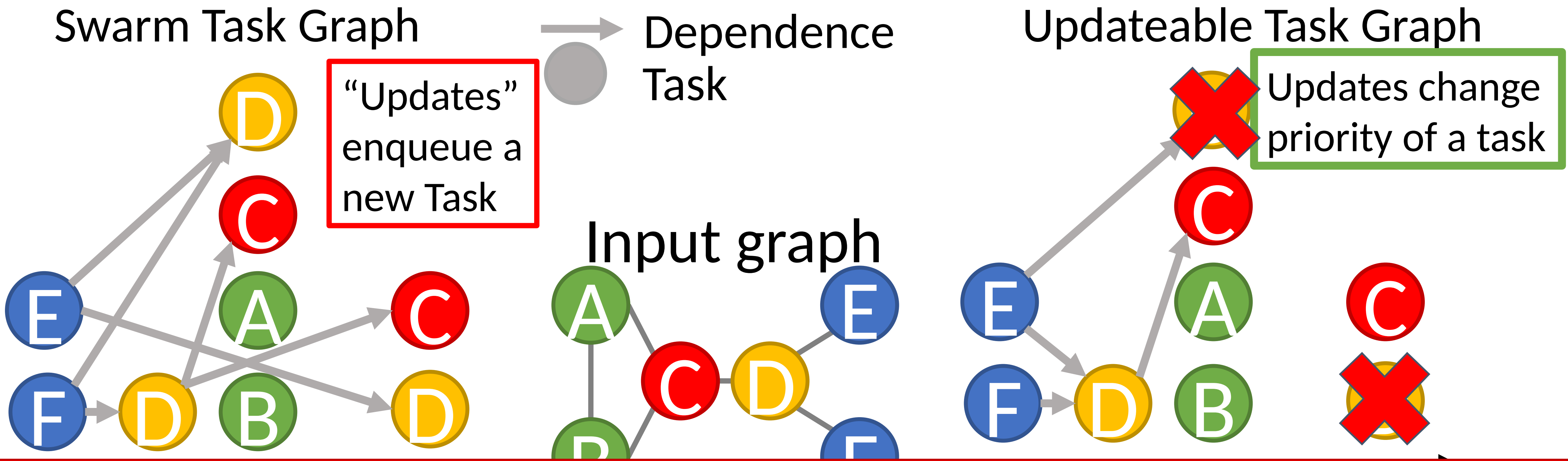
Manual priority tracking

Early exit for **moot** tasks

Tasks that exit early are **moot**: they might as well not run at all

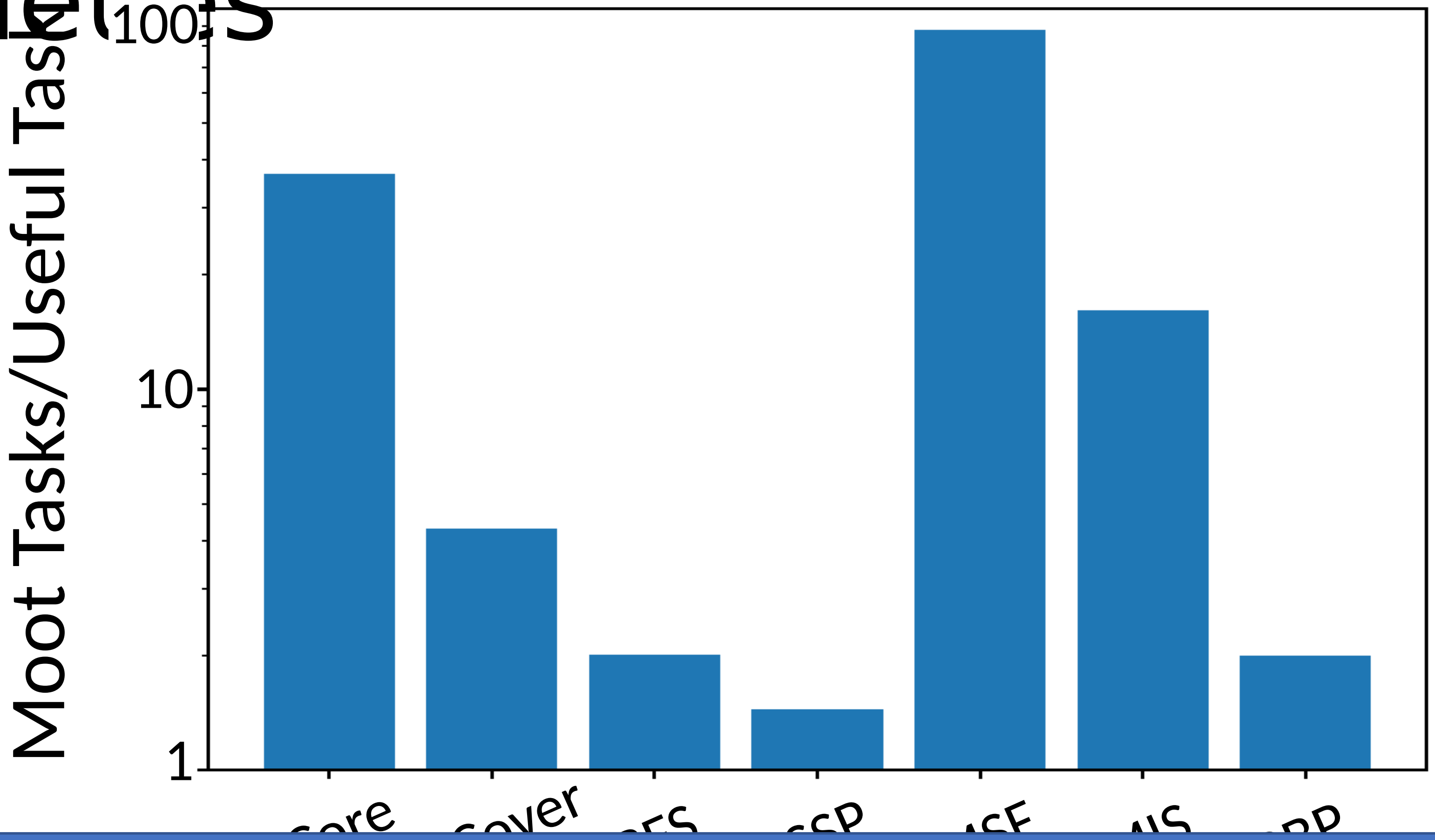
Updateable schedules are efficient

Enqueue-only schedule has 3 more tasks than updateable schedule



Swarm runs Moot tasks, but they might as well not run at all

Moot tasks outnumber useful dequeues



Most tasks are **moot** (useless work in Swarm)

The Hive Execution Model

Understanding Hive tasks and objects

```
void removeV(int v, Timestamp ts) {
  coreness[v] = ts;
  for (int nbr : G.edges[v]) {
    Timestamp prev = hive::getTS(nbr);
    if (prev > ts)
      hive::update(&removeV, nbr, prev-1);
}}
```

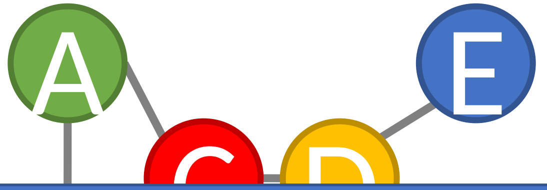
Object

Timestamp

Task Function

Object Table

2	A
2	B
3	C
3	D
1	E
1	F



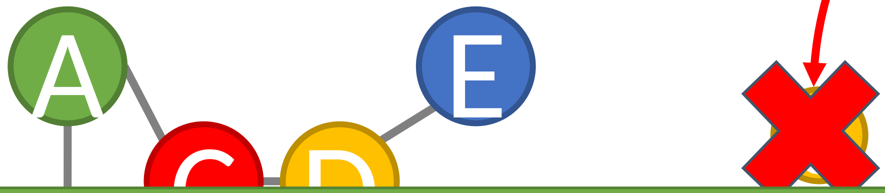
Update binds a task to an object and schedules it to run

Updating an occupied Hive object

```
void removeV(int v, Timestamp ts) {
  coreness[v] = ts;
  for (int nbr : G.edges[v]) {
    Timestamp prev = hive::getTS(nbr);
    if (prev > ts)
      hive::update(&removeV, nbr, prev-1);
}}
```

Object Table

2	A
2	B
3	C
2	D
1	E
1	F



Hive doesn't waste time or space on **moot** tasks

Hive supports many programming patterns

Benchmark	Increment	UpdateMin	Cancel	Update
KCore				
Set Cover				
Astar				
Breadth First Search				
SSSP				
Minimum Spanning Forest				
Maximal Independent Set				
Maximal Matching				
Residual Belief Propagation				

No Priority Queue in Sequential Implementation

Parallelizing Priority Updates

Hive speculates to run tasks in parallel

For each task, Hive speculates that:

- Eager data speculation: Predecessors have already performed their writes

- Eager control speculation: Its parent will not abort

The same as Swarm [Jeffrey et al. MICRO'15]

- Eager scheduler speculation: It will not be replaced by an update

Priority updates are scheduler dependences

- The scheduler dependence is old
  - Found in self-modifying code [Wilkes and Renwick. '49]
- Created by priority updates
  - When a task replaces a later-scheduled task, it creates a scheduler dependence
- Can be predicated into data and control dependences
  - Moot tasks are like predicated instructions in straight-line code

```
STR R5, [PC, #4]
ADD R1, R1, R1
```

Updates have a different dependence, they need different speculation

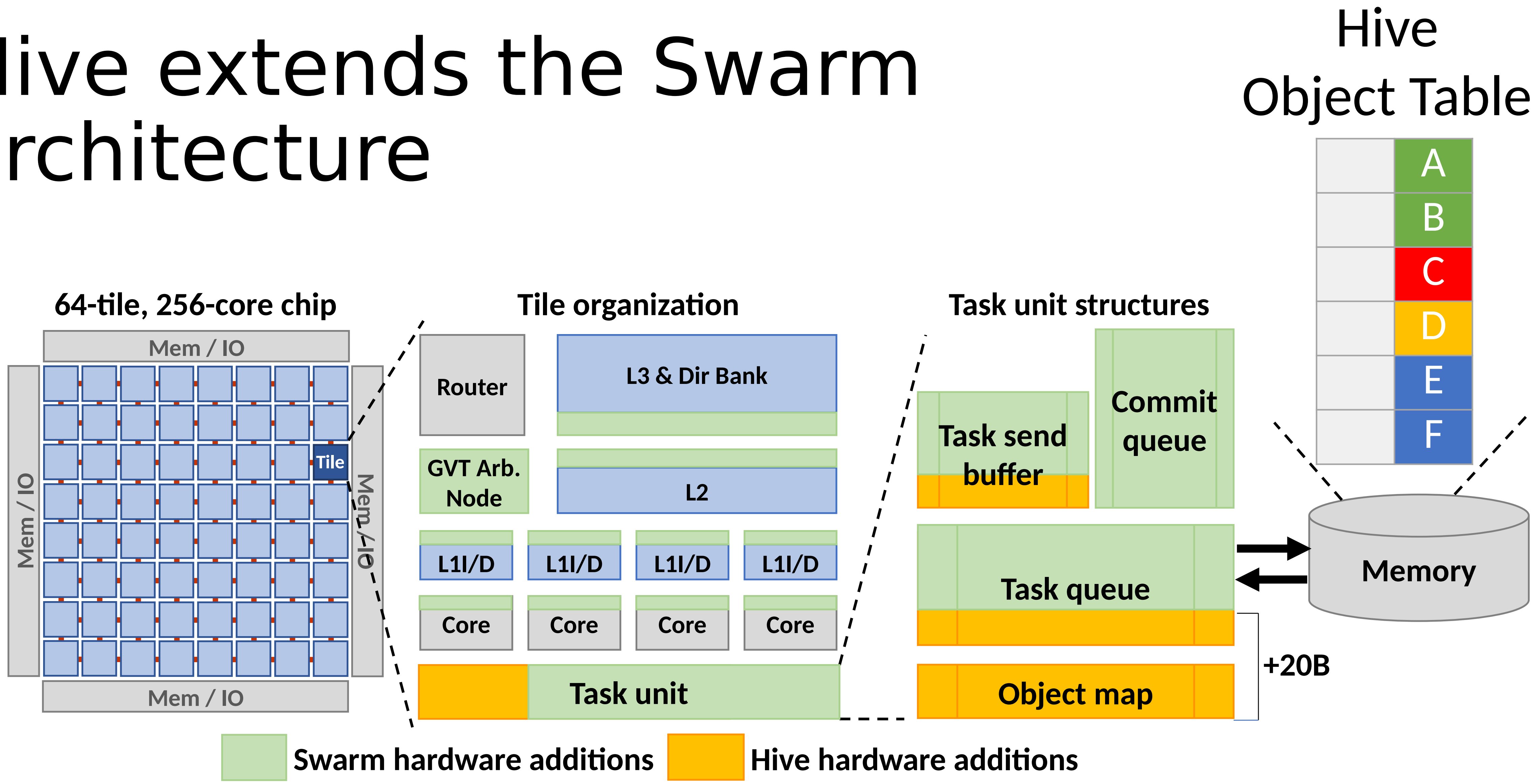
Scheduler speculation:
Task versioning and Mootness detection

- Maintain multiple versions of each task
  - 1 for each speculative update + up to 1 non-speculative
- 1 task version is speculatively valid, all others are speculatively Moot
  - Speculatively Moot task versions are not runnable
- When Mootness becomes non-speculative, discard the Moot version

- Mootness can detected by comparing timestamps of parents

Hive avoids running moot tasks and reduces their speculative state

Hive extends the Swarm architecture



9% Task Unit Area Increase

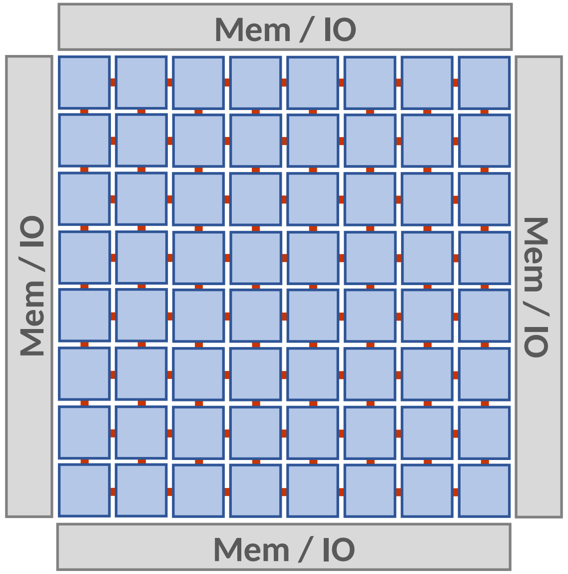
3% Area of a Nehalem Processor

Evaluation

Methodology

Event-driven, Pin-based Simulator

64 Tiles, 256 Cores



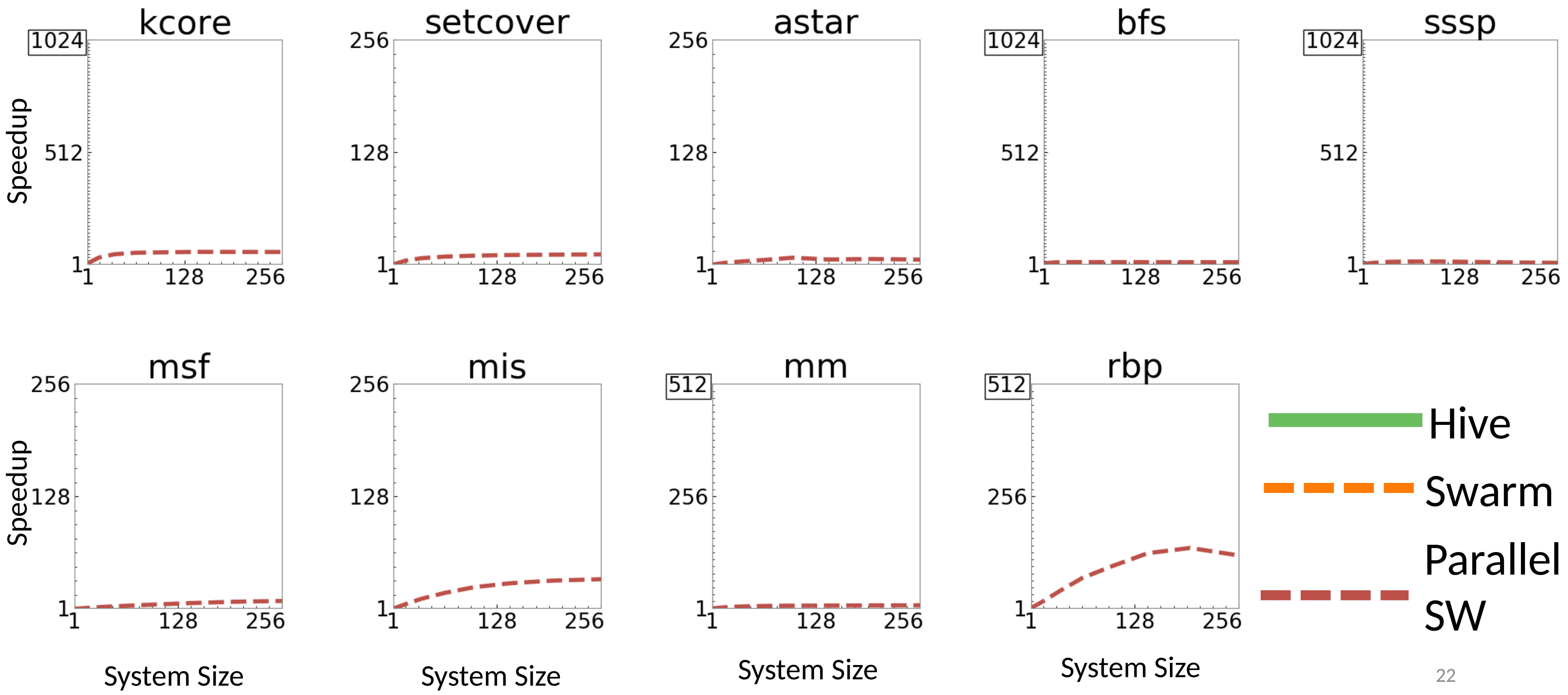
32kB L1 per core
1MB L2 per tile
256MB LLC
4 In-order, single-issue scoreboarded cores/tile
64 Task Queue entries/core
16 Commit Queue entries/core

Scalability experiments up to 256 cores

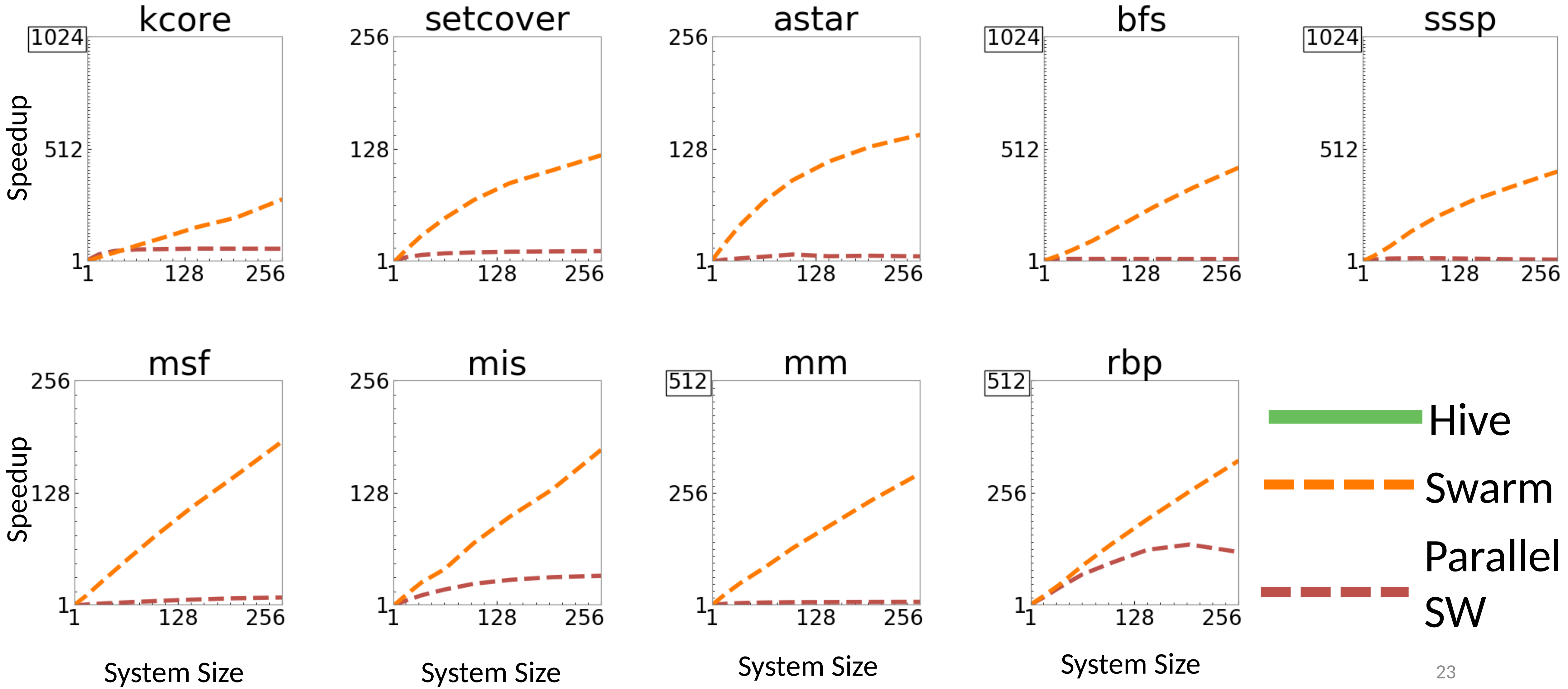
- Smaller systems have fewer tiles

9 applications: KCore, Setcover, astar, BFS, SSSP, MSF, MIS, MM, RBP

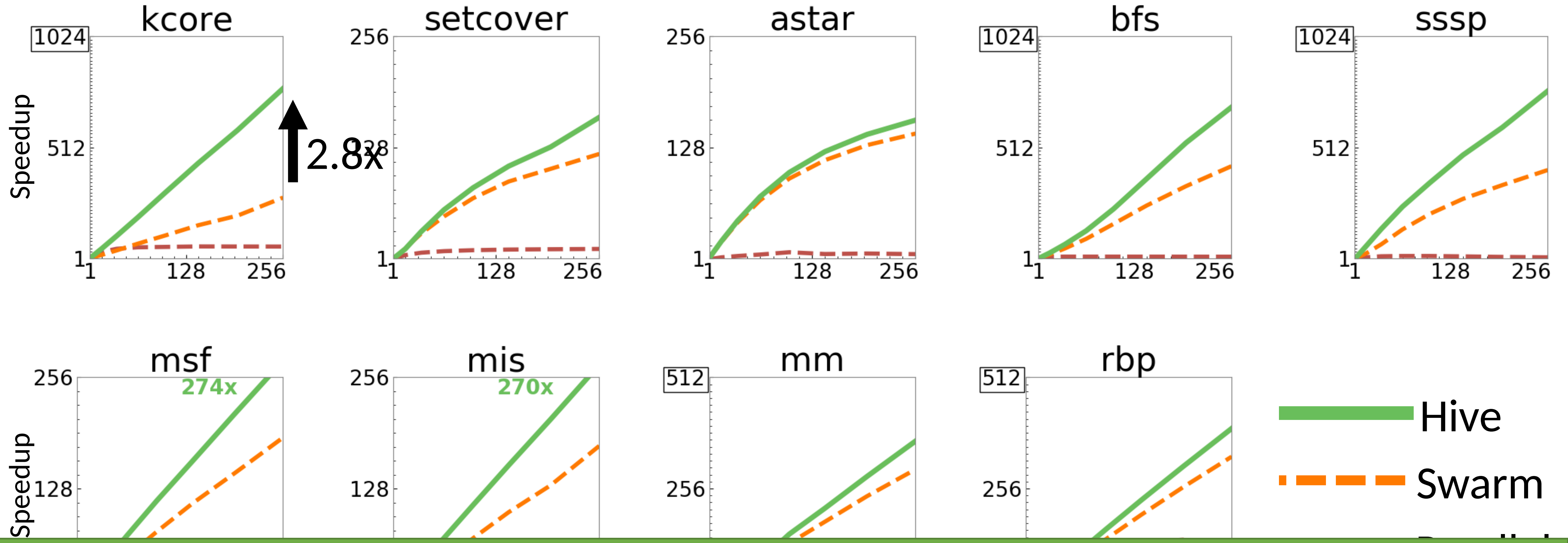
Software struggles to scale beyond 100c



Swarm scales well sometimes



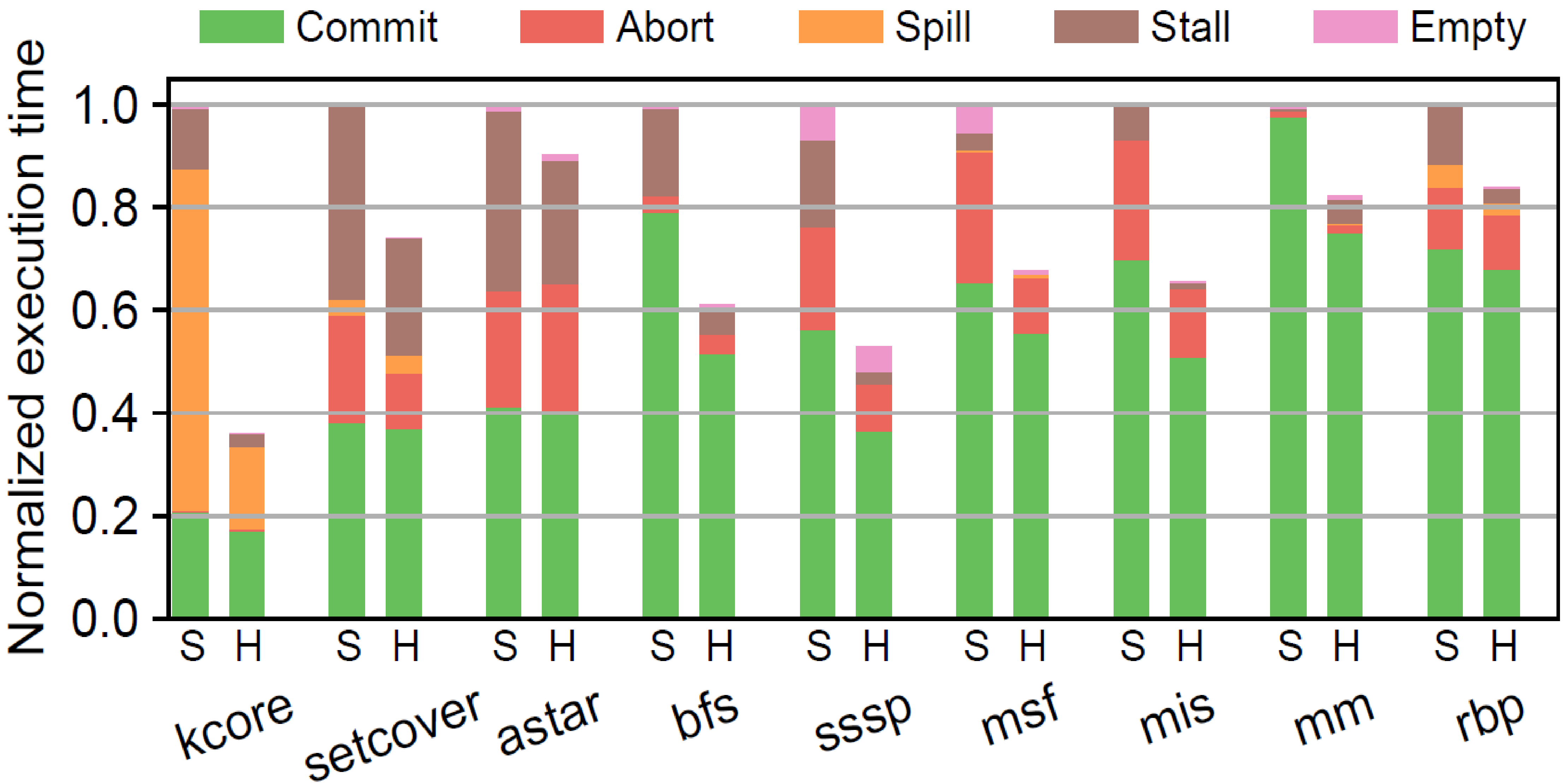
Hive is faster than Swarm



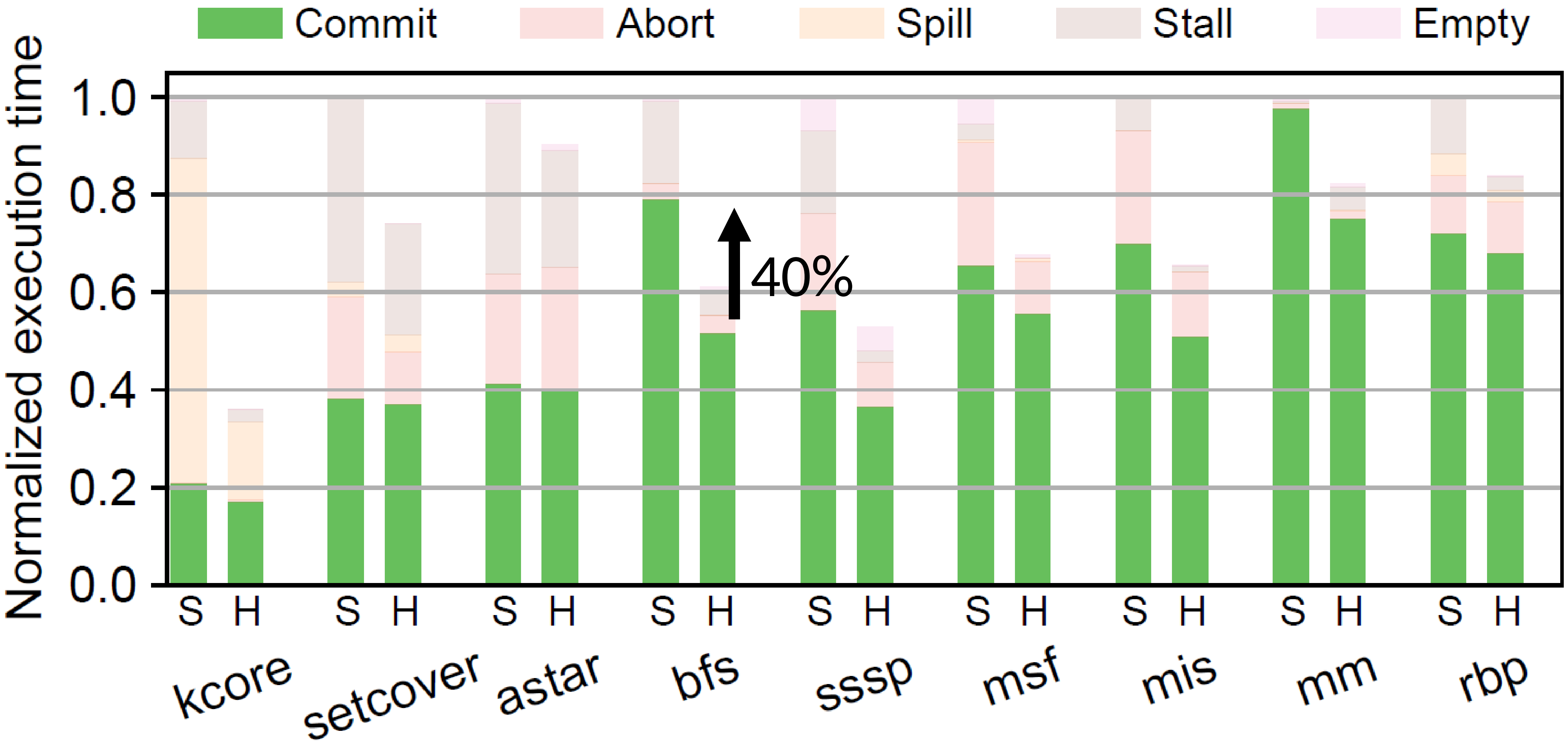
Hive is up to 2.8x faster than Swarm

System Size System Size System Size System Size

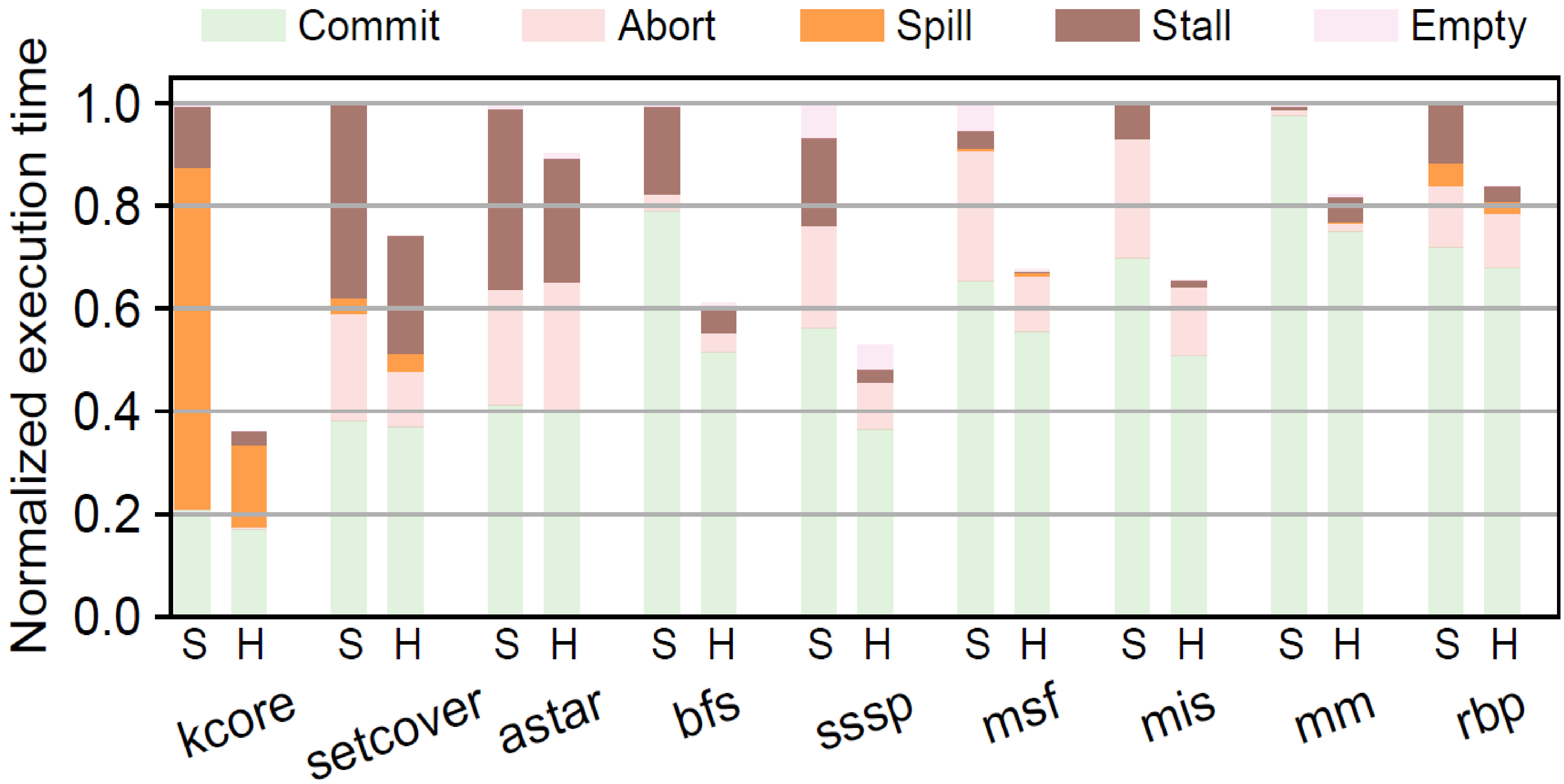
Breaking down Hive vs. Swarm at 256 cores



Hive does less work



Hive reduces queue pressure



Conclusions and Q+A

- Priority updates are useful operations for ordered algorithms
- The scheduler dependences created by these updates require task versioning and mootness detection for speculation
- Hive extracts parallelism by speculating on data, control, and scheduler dependences

Gilead Posluns, Yan Zhu, Guowei Zhang, Mark C. Jeffrey

ISCA 2022

UNIVERSITY OF TORONTO
HUAWEI

Extra Slides

Implementing KCore With Only Enqueues

```
PriorityQueue pq;
for (int v: G.V)
  pq.enqueue(v, G.degree[v]);
while (!pq.empty()) {
  int v, int prio = pq.dequeueMin();
  coreness[v] = prio;
  for (int nbr : G.edges[v])
    if (pq.getPrio(nbr) > prio)
      pq.decrementPrio(nbr);
}
```

```
void removeV(Timestamp ts, Vertex* v) { // Task

}
```

Implementing KCore With Only Enqueues

```
PriorityQueue pq;
for (int v : G.V)
  pq.enqueue(v, G.degree[v]);
while (!pq.empty()) {
  int v, int prio = pq.dequeueMin();
  coreness[v] = prio;
  for (int nbr : G.edges[v])
    if (pq.getPrio(nbr) > prio)
      pq.decrementPrio(nbr);
}
```

```
void removeV(Timestamp ts, Vertex* v) { // Task

  for (Vertex* ngh : v->neighbors()) {




}
```

Implementing KCore With Only Enqueues

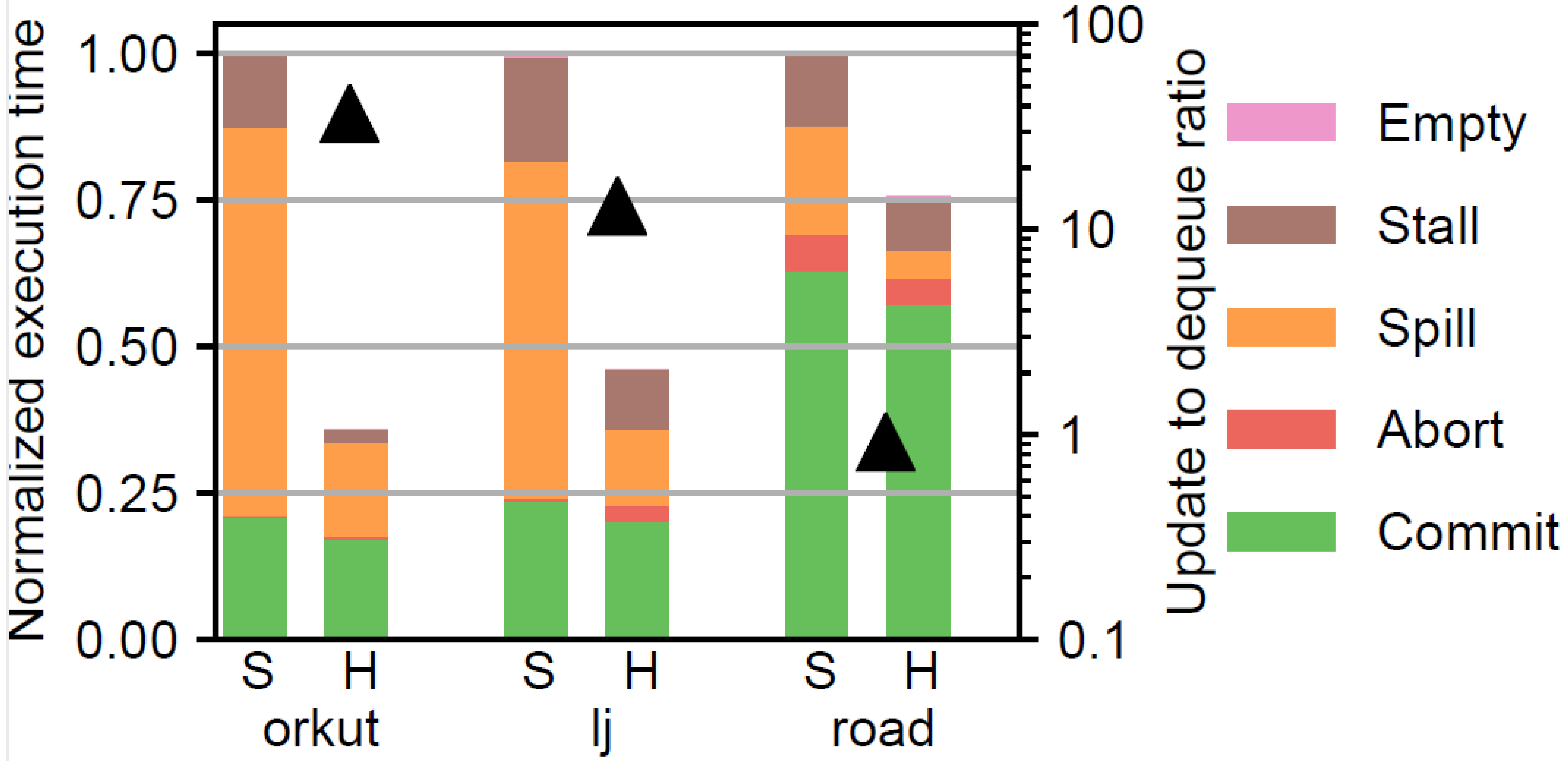
```
PriorityQueue pq;
for (int v : G.V)
  pq.enqueue(v, G.degree[v]);
while (!pq.empty()) {
  int v, int prio = pq.dequeueMin();
  coreness[v] = prio;
  for (int nbr : G.edges[v])
    if (pq.getPrio(nbr) > prio)
      pq.decrementPrio(nbr);
}
```

```
Timestamp prios[G.n]; // Scheduling metadata
void removeV(Timestamp ts, Vertex* v) { // Task

  for (Vertex* ngh : v->neighbors()) {
    if (prios[ngh->id] <= ts) continue;



}
```

Implementing KCore With Only Enqueues

```
PriorityQueue pq;
for (int v : G.V)
  pq.enqueue(v, G.degree[v]);
while (!pq.empty()) {
  int v, int prio = pq.dequeueMin();
  coreness[v] = prio;
  for (int nbr : G.edges[v])
    if (pq.getPrio(nbr) > prio)
      pq.decrementPrio(nbr);
}
```

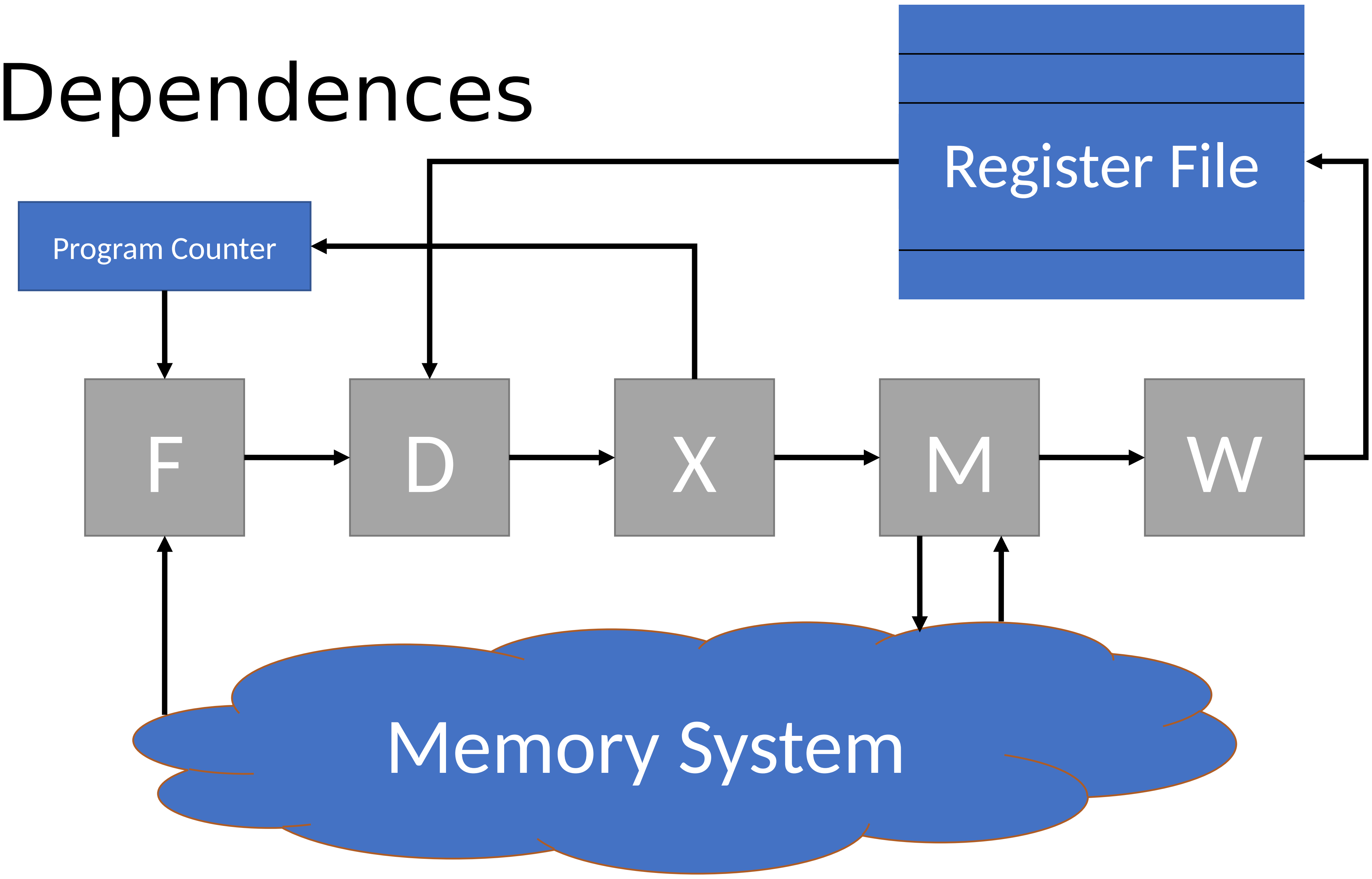
```
Timestamp prios[G.n]; // Scheduling metadata
void removeV(Timestamp ts, Vertex* v) { // Task
  if (prios[v->id] < ts) return;
  for (Vertex* ngh : v->neighbors()) {
    if (prios[ngh->id] <= ts) continue;
    prios[ngh->id]--;
    swarm::enqueue(&removeV, prios[ngh->id], ngh);
  }
}
```


Normalized execution time
1.00
0.75
0.50
0.25
0.00
S
H
S
H
S
H
orkut
lj
road
100
10
1
0.1
Update to dequeue ratio
Empty
Stall
Spill
Abort
Commit

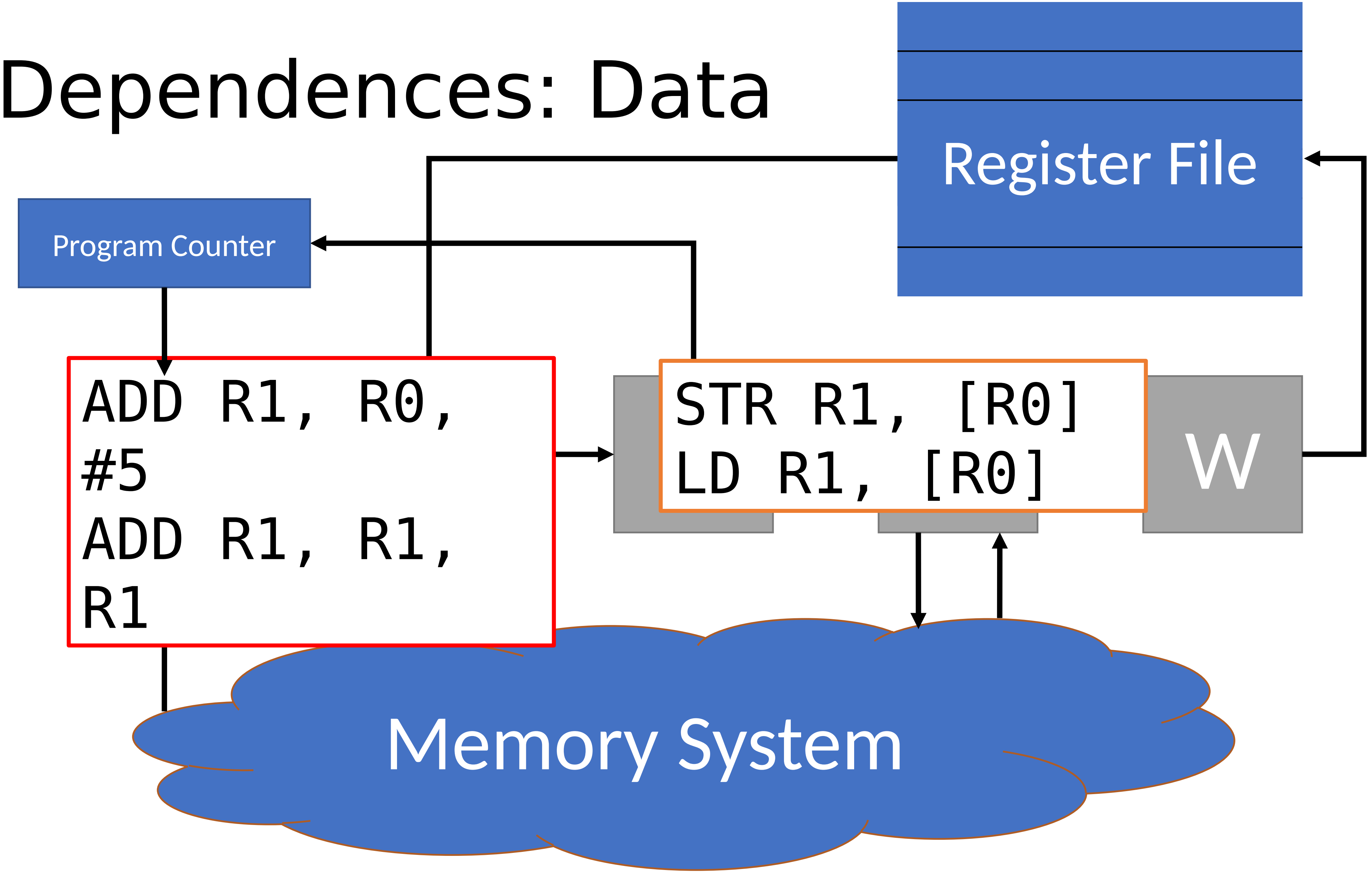
Benchmark	**Input**	**SW parallel strategy**	**Hive programming pattern(s)** (Sec. 3.3)	**1-core cycles**	**Avg. task length**		**1-core vs. serial**	
					Swarm	Hive	Swarm	Hive
kcore [23]	com-Orkut [82]	Bulk-Synchronous	Incremental	219B	364	241	0.42×	1.09×
setcover [23]	com-Orkut [82]	Relaxed PQ	UpdateMin and Postpone	14.6B	137	140	2.47×	1.53×
astar [37]	Germany roads [2]	Relaxed PQ [51, 56]	UpdateMin	1.4B	848	1056	1.22×	1.36×
bfs [45]	hugetric-00020 [12, 22]	Bulk-Synchronous	UpdateMin	3.2B	117	150	0.76×	0.96×
sssp [58]	East USA roads [1]	Relaxed PQ [51, 56]	UpdateMin	2.0B	246	258	1.65×	2.24×
msf [69]	kronecker_log16n[12, 22]	Speculation [14]	UpdateMin and Cancel	0.50B	137	257	2.24×	3.08×
mis [69]	R-MAT [19]	Speculation [14, 15]	Cancel	2.0B	119	81	0.68×	0.98×
mm [69]	com-Orkut [82]	Speculation [14, 15]	Cancel	22.5B	147	147	0.53×	0.51×
rbp [4]	200×200 Ising [20]	Relaxed PQ [64]	Update	18.4B	1128	1697	0.90×	1.23×

The Scheduler Dependence

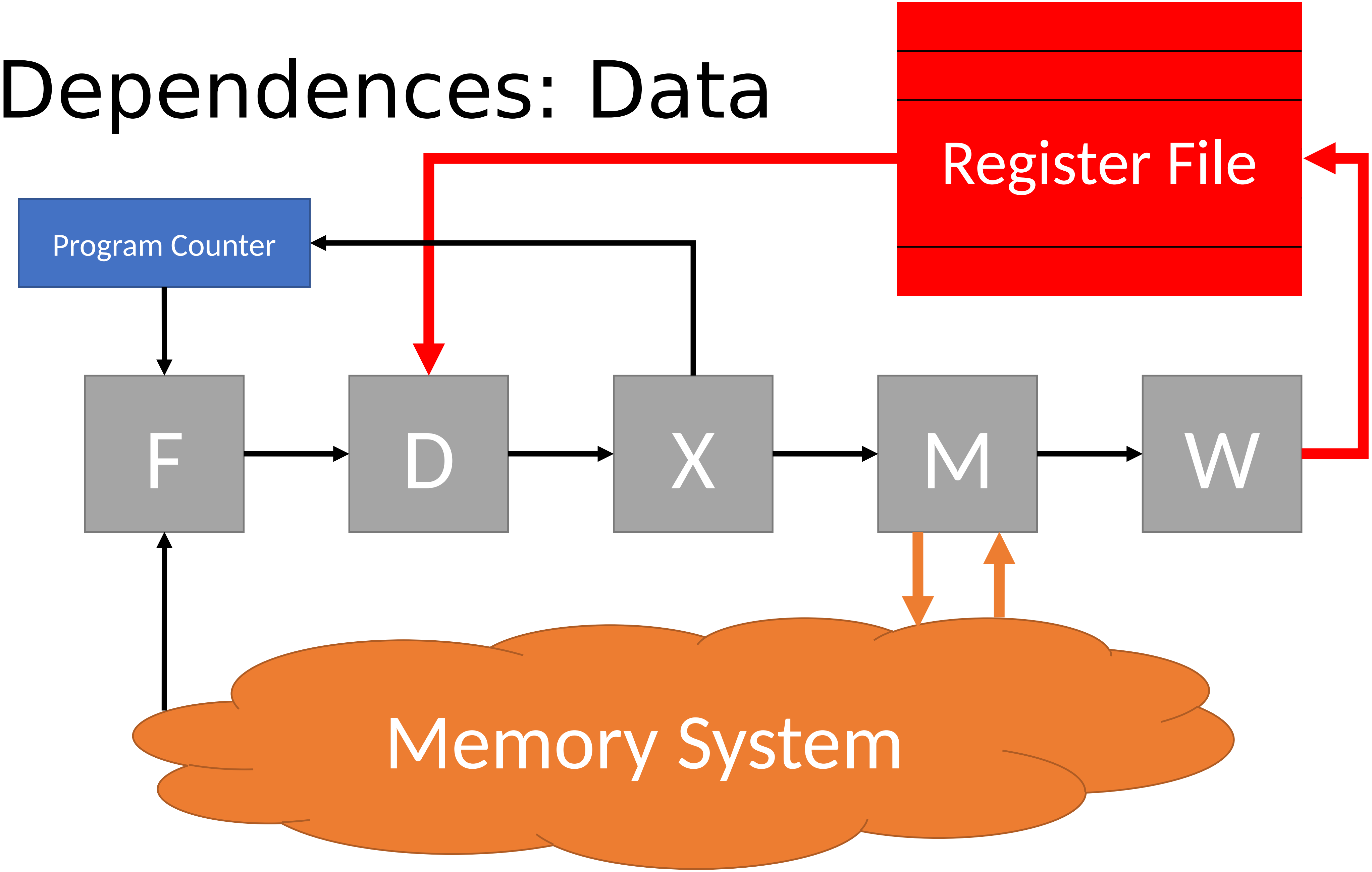
Dependences



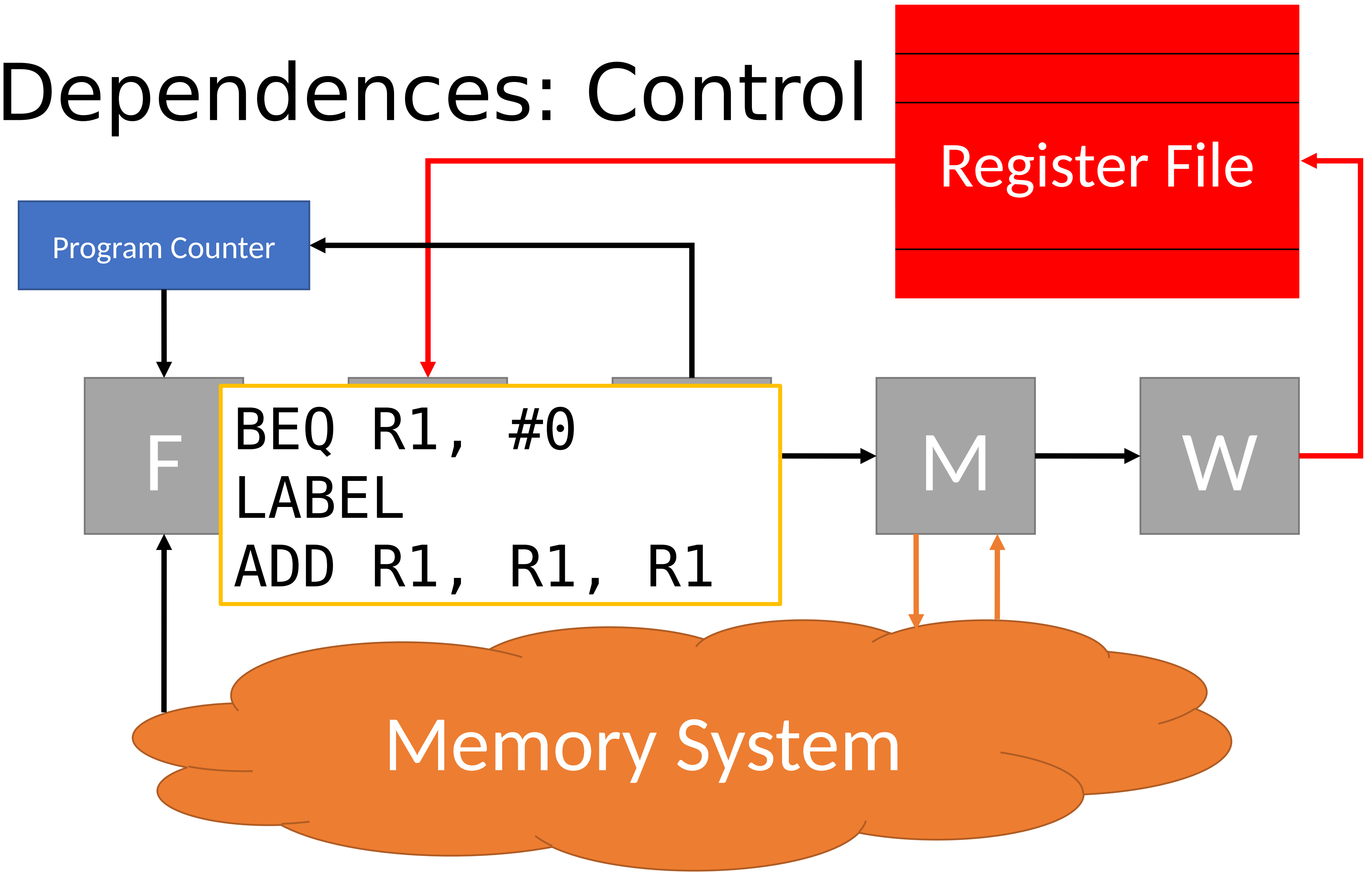
Dependences: Data



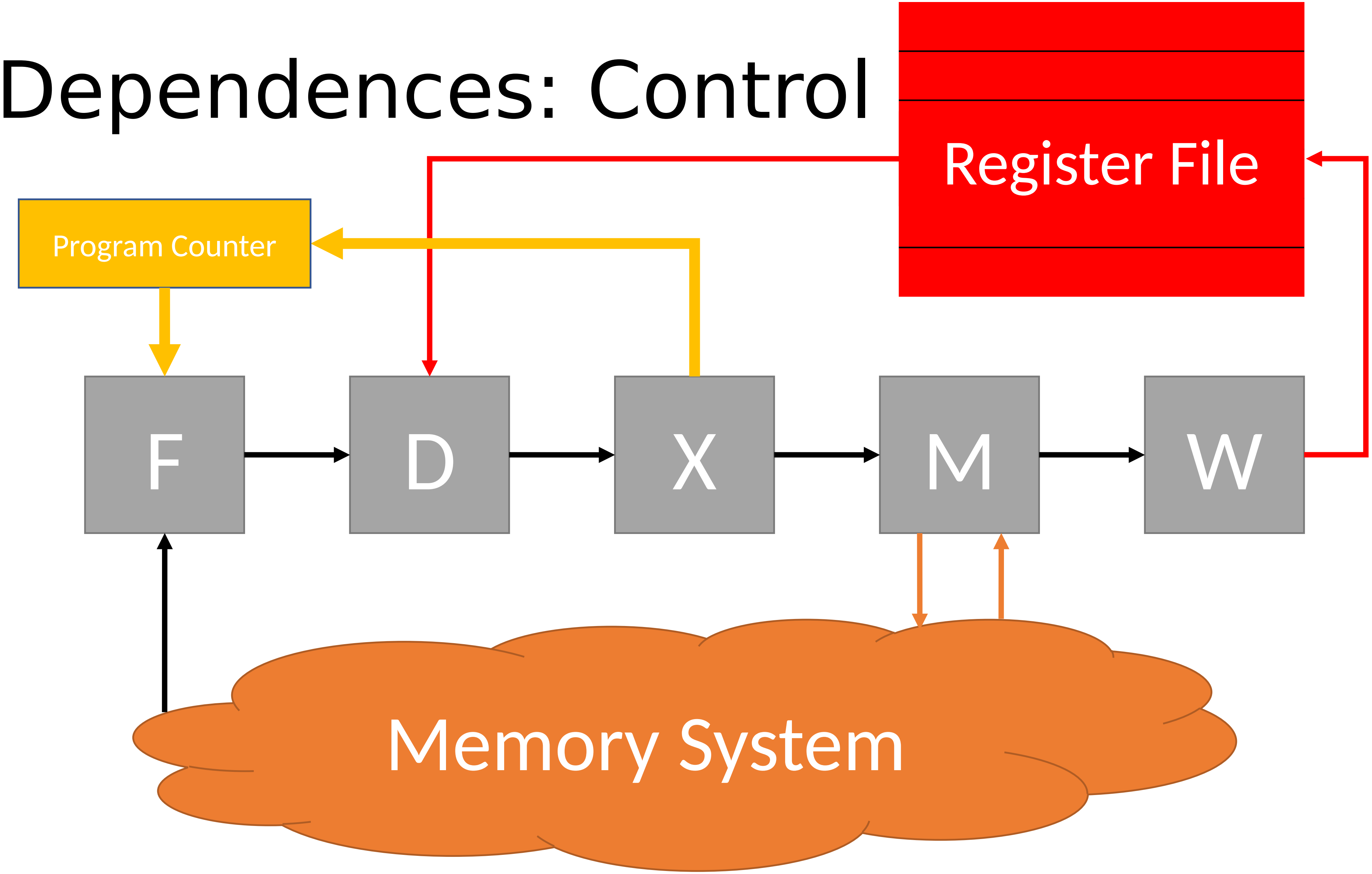
Dependences: Data



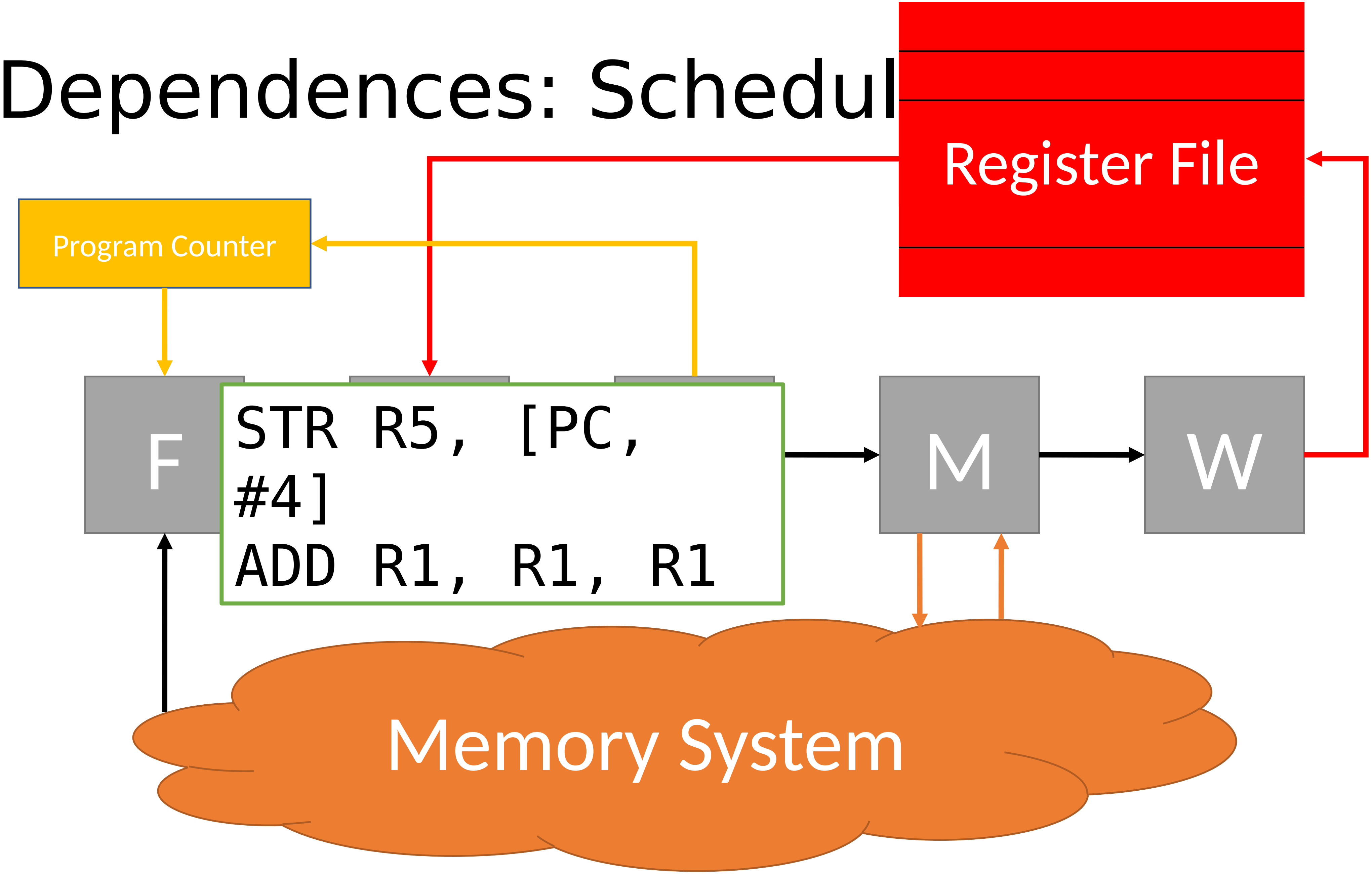
Dependences: Control



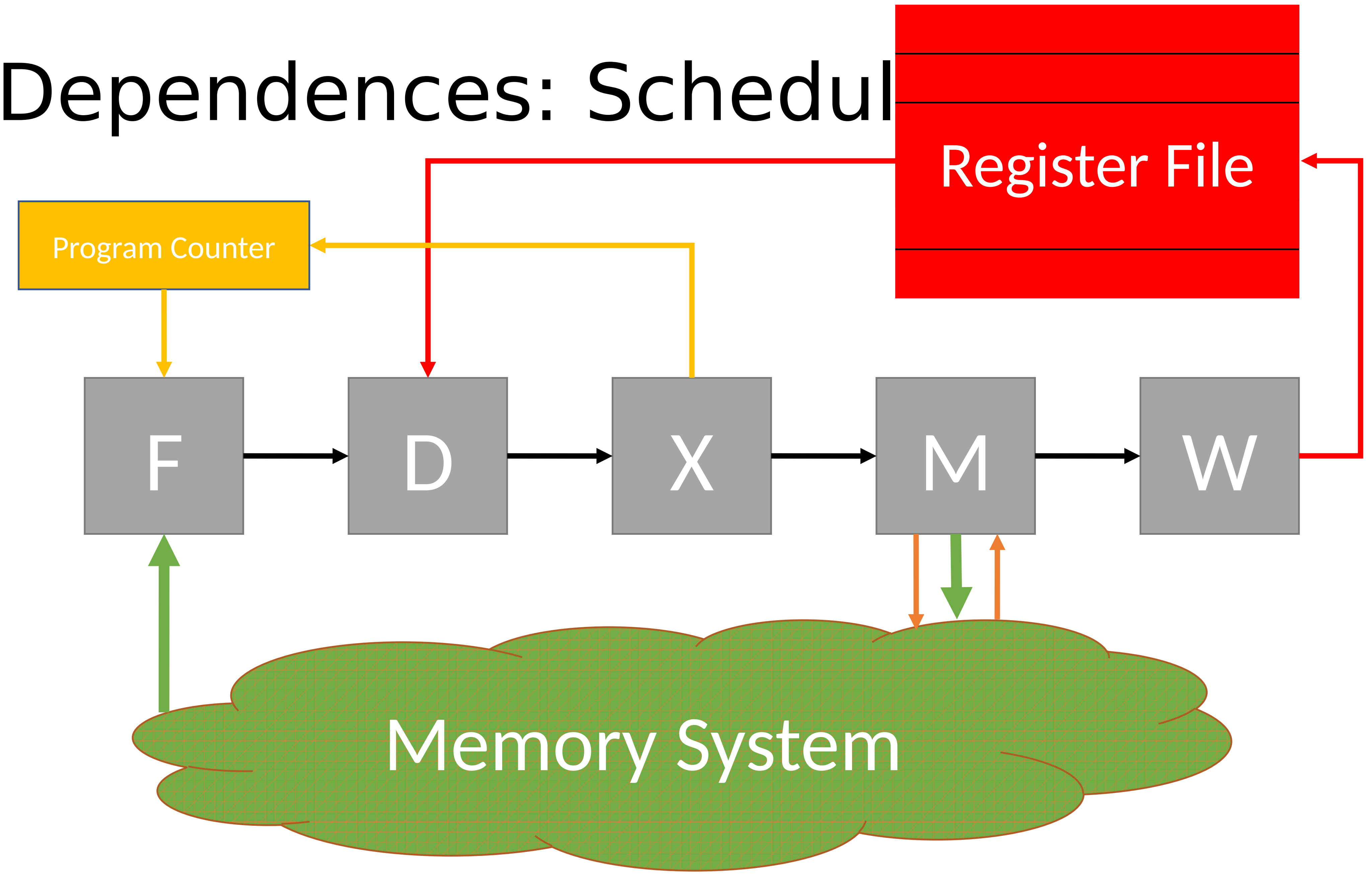
Dependences: Control



Dependences: Schedul

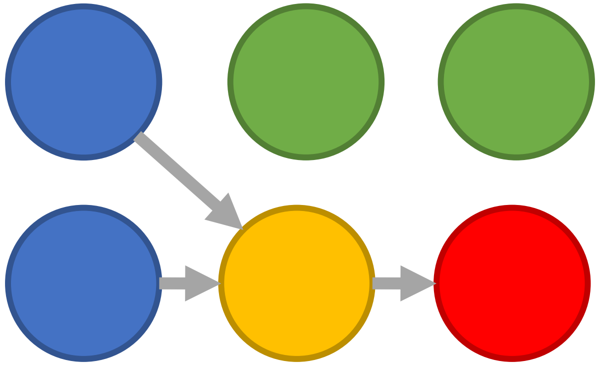


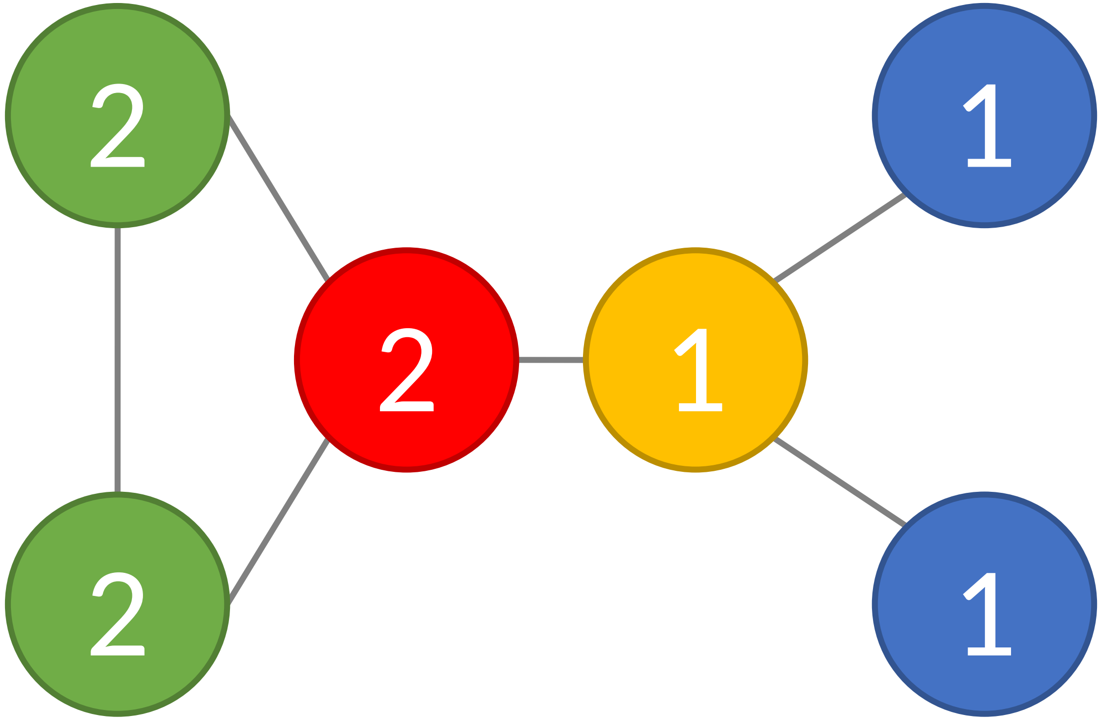
Dependences: Schedul



Where's the parallelism in Kcore?

```
PriorityQueue pq;
for (int v: G.V)
  pq.enqueue(v, G.degree[v]);
while (!pq.empty()) {
  int v, int prio = pq.dequeueMin();
  coreness[v] = prio;
  for (int nbr : G.edges[v])
    if (pq.getPrio(nbr) > prio)
      pq.decrementPrio(nbr);
}
```

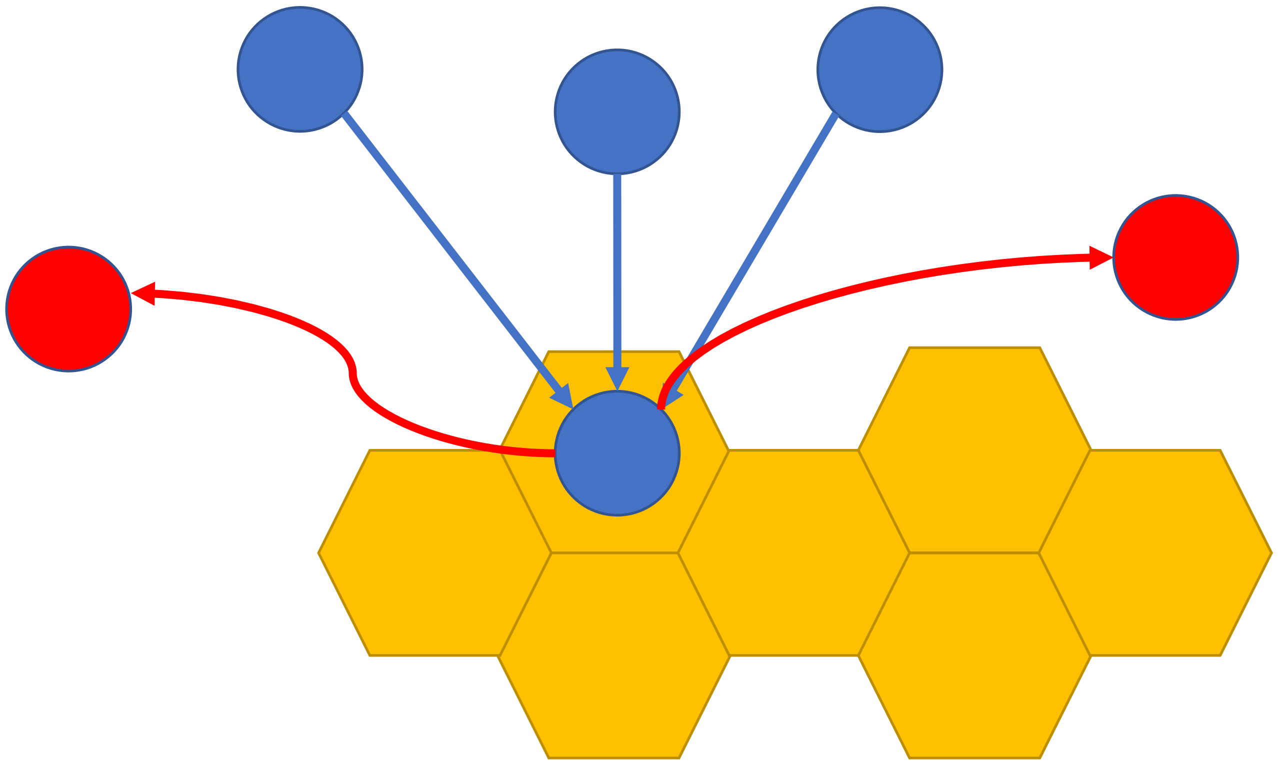




Hive Speculates on Update Operations

Hive speculates that all `update` calls will commit.

Hive uses *eager* **Task** versioning.



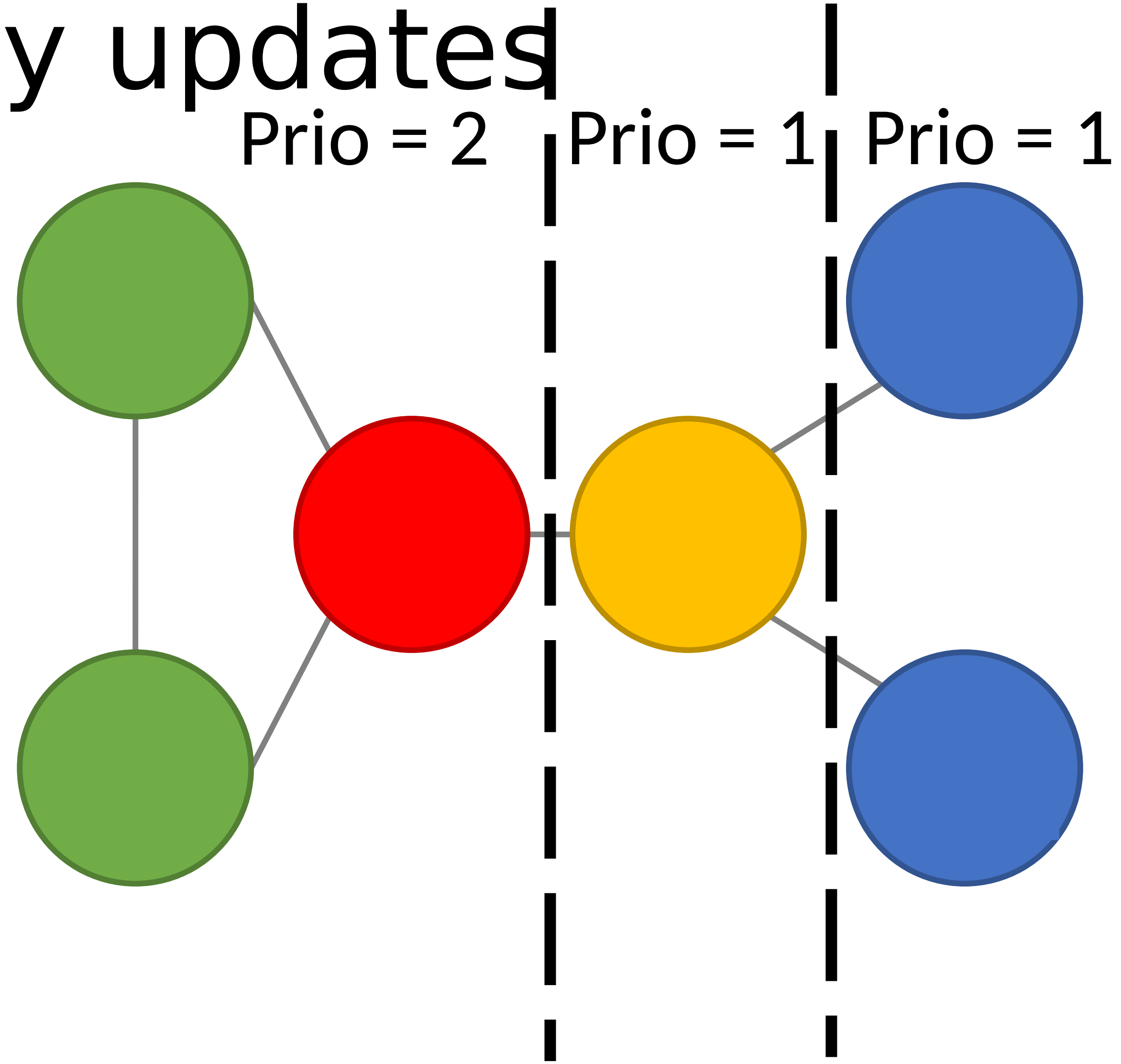
Detecting scheduler dependences

- Each new **Task** version is sent to a tile based on its **Object**
- When a Task Unit receives a new **Task** version, it checks for other versions with the same **Object**
- 2 **Task** versions with the same **Object** are a potential Scheduler Dependence
- The Task Unit compares the VTs of the **Task**s and their parents to determine which, if any, is Moot.

Scheduler Dependencies Are Cheap To Detect and Handle

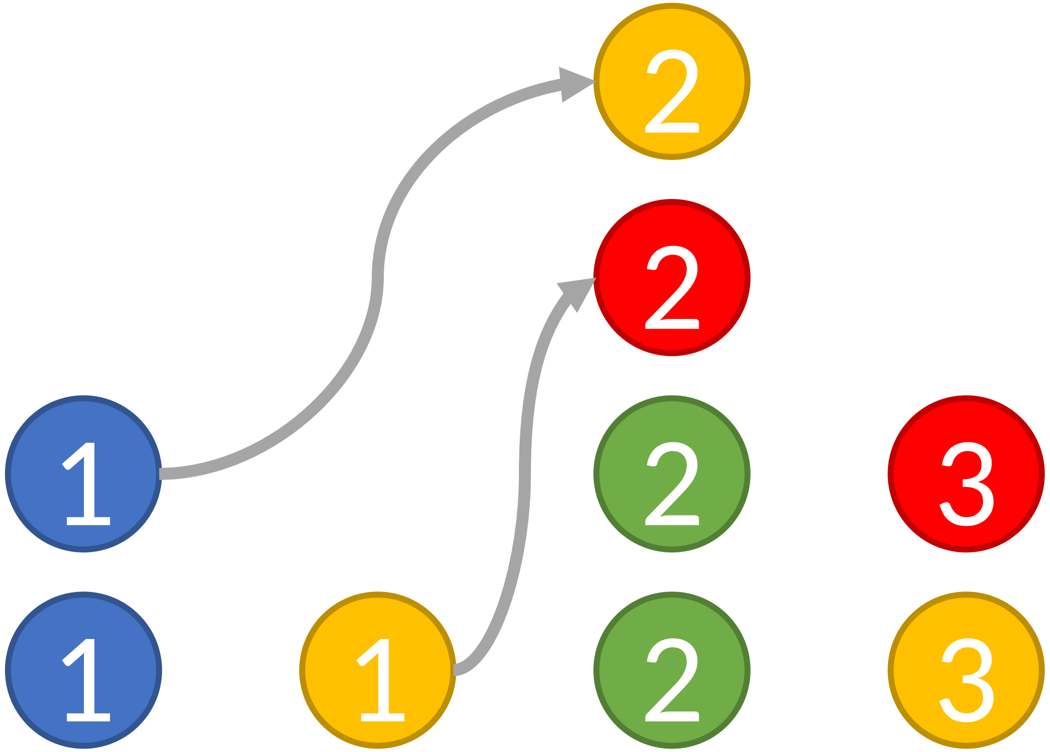
KCore uses priority updates

```
PriorityQueue pq;
for (int v: G.V)
  pq.enqueue(v, G.degree[v]);
while (!pq.empty()) {
  int v, int prio = pq.dequeueMin();
  coreness[v] = prio;
  for (int nbr : G.edges[v])
    if (pq.getPrio(nbr) > prio)
      pq.decrementPrio(nbr);
}
```



The Swarm Architecture [Jeffrey et al, MICRO '15]

Task-Based Execution Model



Hive tasks and objects

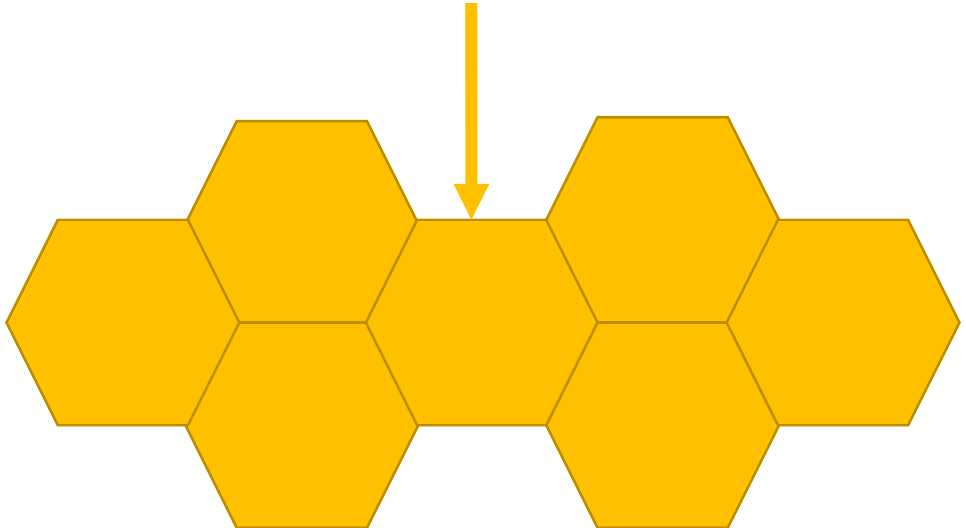
- A Hive program is composed of **Task**s and **Object**s
- **Task**s are like Swarm tasks
- **Object**s are how a Hive program queries and updates tasks

```
hive::update(
        fn,  //what to do
        oid, //what to do it to
        ts,  //when to do it
        args //what to do it with);
```

Hive objects contain tasks

- Hive **Objects** are containers for **Tasks**
  - An **Object** contains up to one **Task**, or a Timestamp where a **Task** used to be
- Hive Programs call `init` to create a set of objects, which are initially empty
- Once created, **Objects** are identified by an object ID (oid) from 0 to n

```
hive::init<flags>(n /*number of objects to create*/)
```



Binding Hive tasks to objects

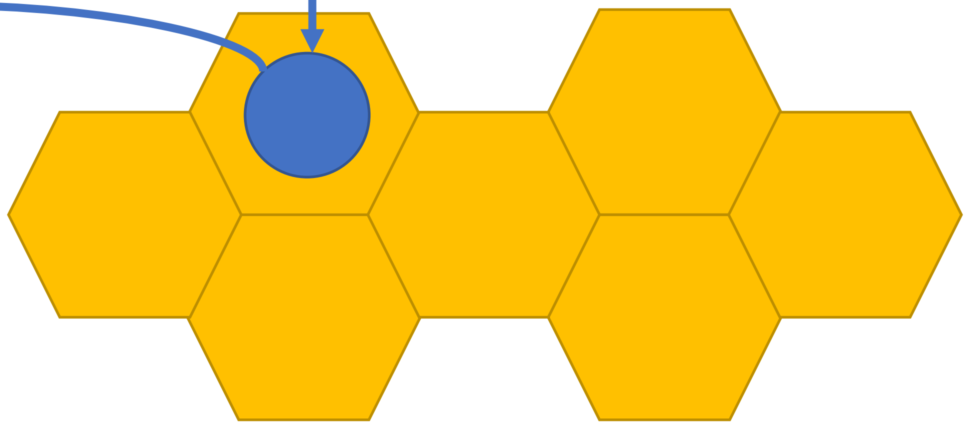
- Instead of `enqueue`, Hive programs can call `update` to bind a **Task** to an **Object**.
- Hive programs can also call `getTS`

  to retrieve the Timestamp

  of the **Task** last bound to

  an **Object**

```
hive::update(
          fn,   //what to do
          oid,  //what to do it to
          ts,   //when to do it
          args  //what to do it with);
```

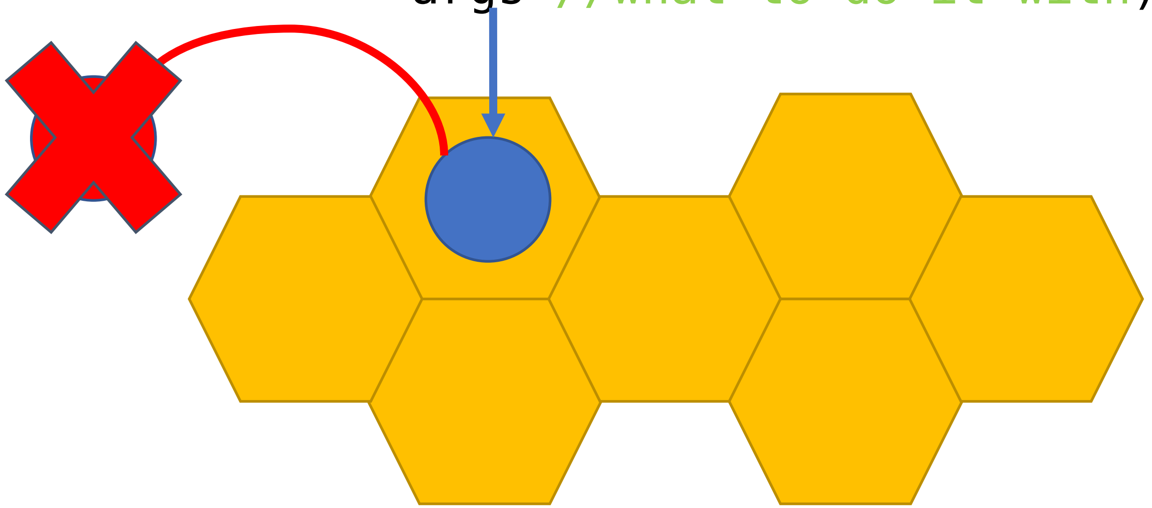


```
Timestamp ts = getTS(oid);
```

Updating a Hive task using its object binding

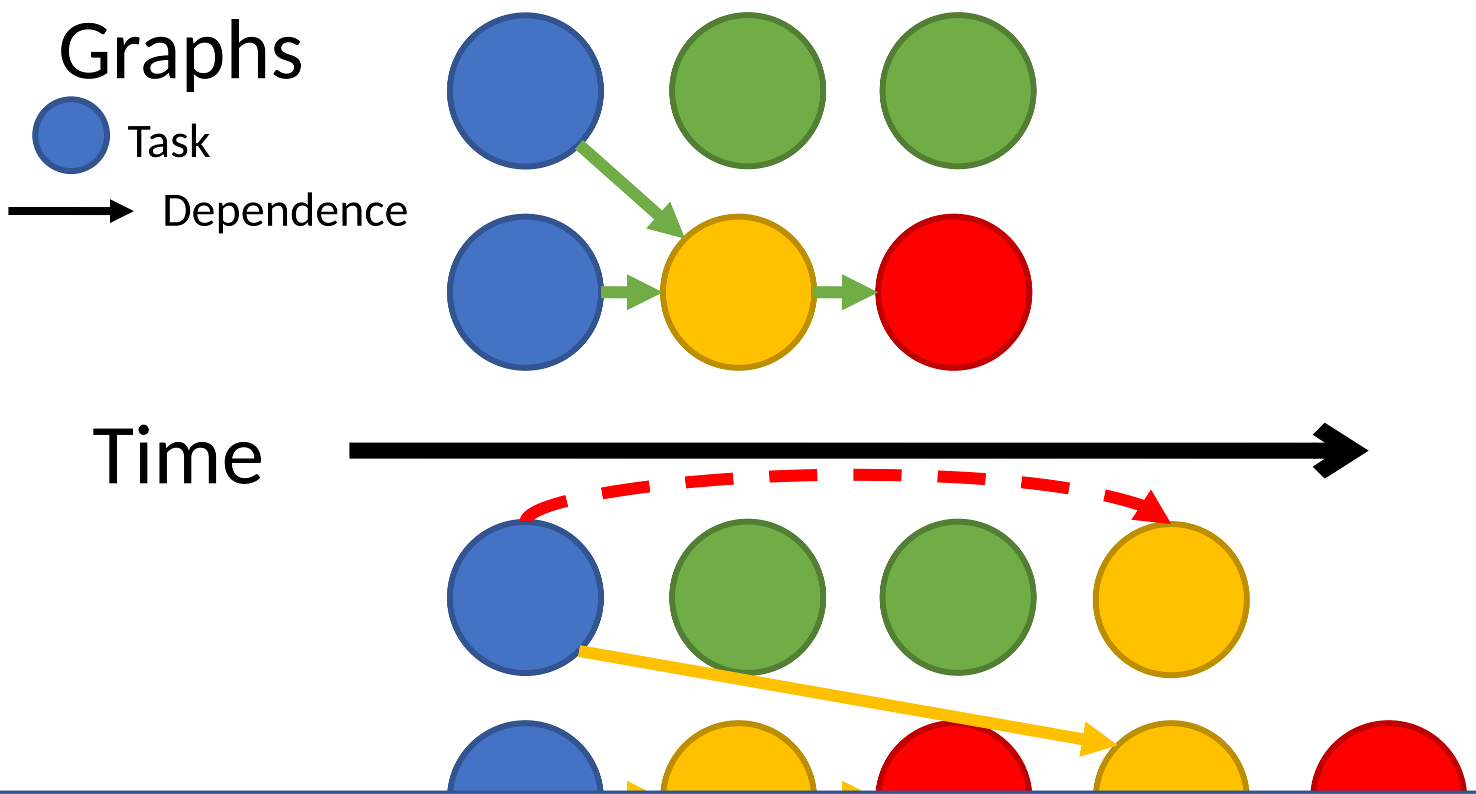
- When a Hive program calls `update` on an **Object** with a **Task** already bound to it
  - The old **Task** is replaced by the new one
  - The old **Task** is destroyed
- This allows a Hive program to `update` a **Task** many times, but only execute it once.

```
hive::update(
        fn,   //what to do
        oid,  //what to do it to
        ts,   //when to do it
        args  //what to do it with);
```



The scheduler dependence is distinct

Task Graphs



Data

```
ADD R0, #5✉R1
ADD R1,
```

```
BEQ R1, #0✉LABEL
ADD R1, R1✉R1
```

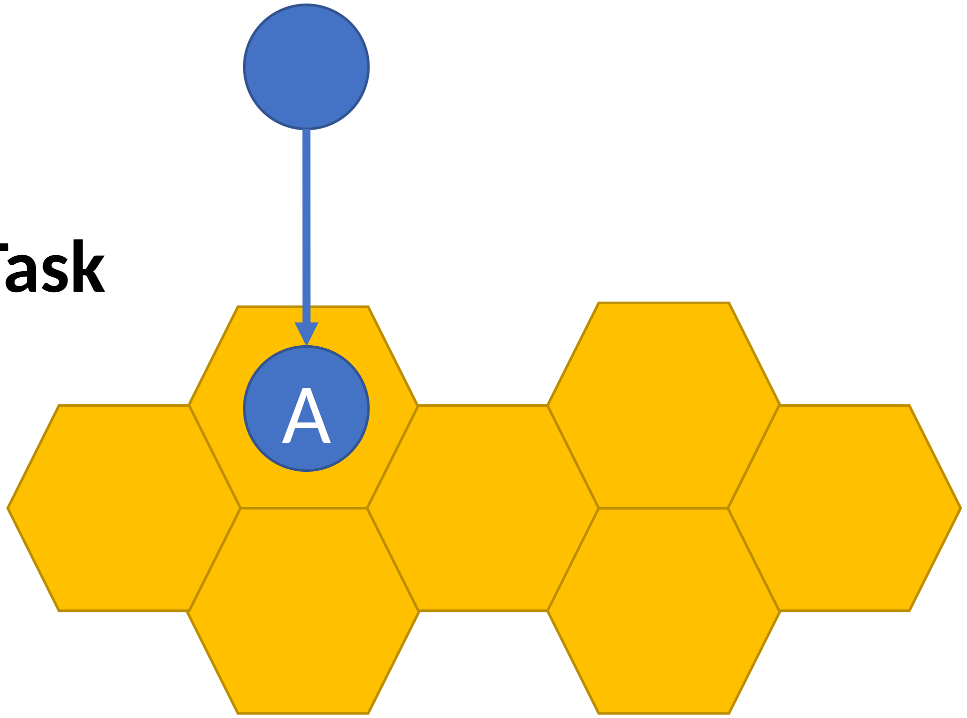
```
STR R5✉[PC, #4]
```

Converting Scheduler Dependences to Data/Control Is Like Predication

Hive recognizes speculatively moot tasks

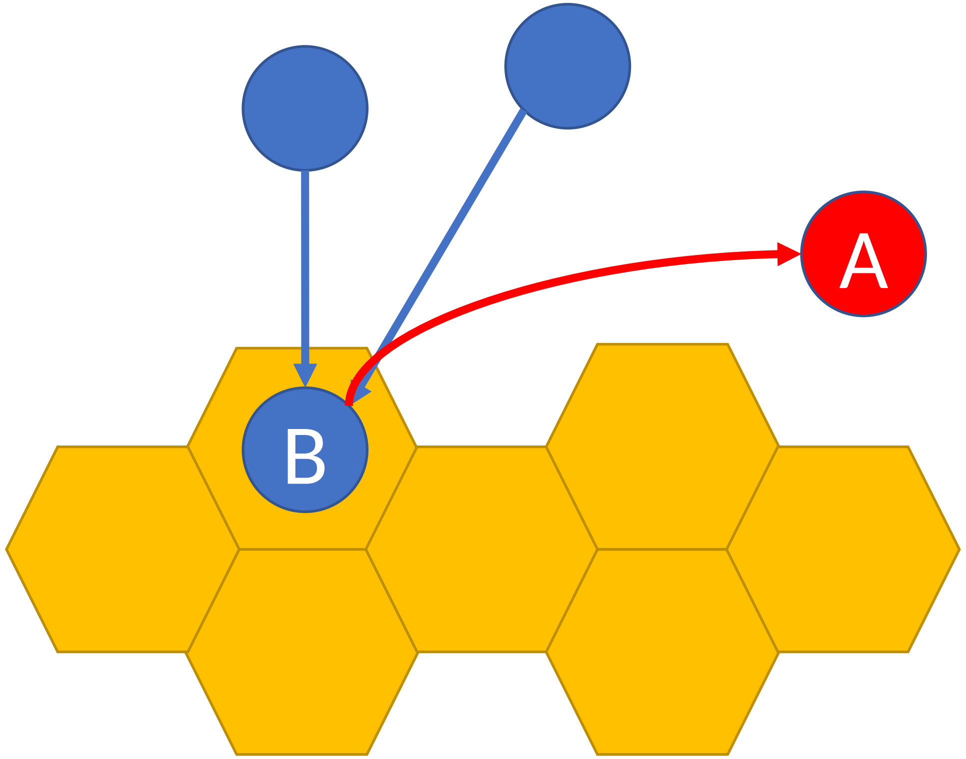
Hive speculates that all `update` calls will commit.

Hive uses *eager* **Task** versioning.



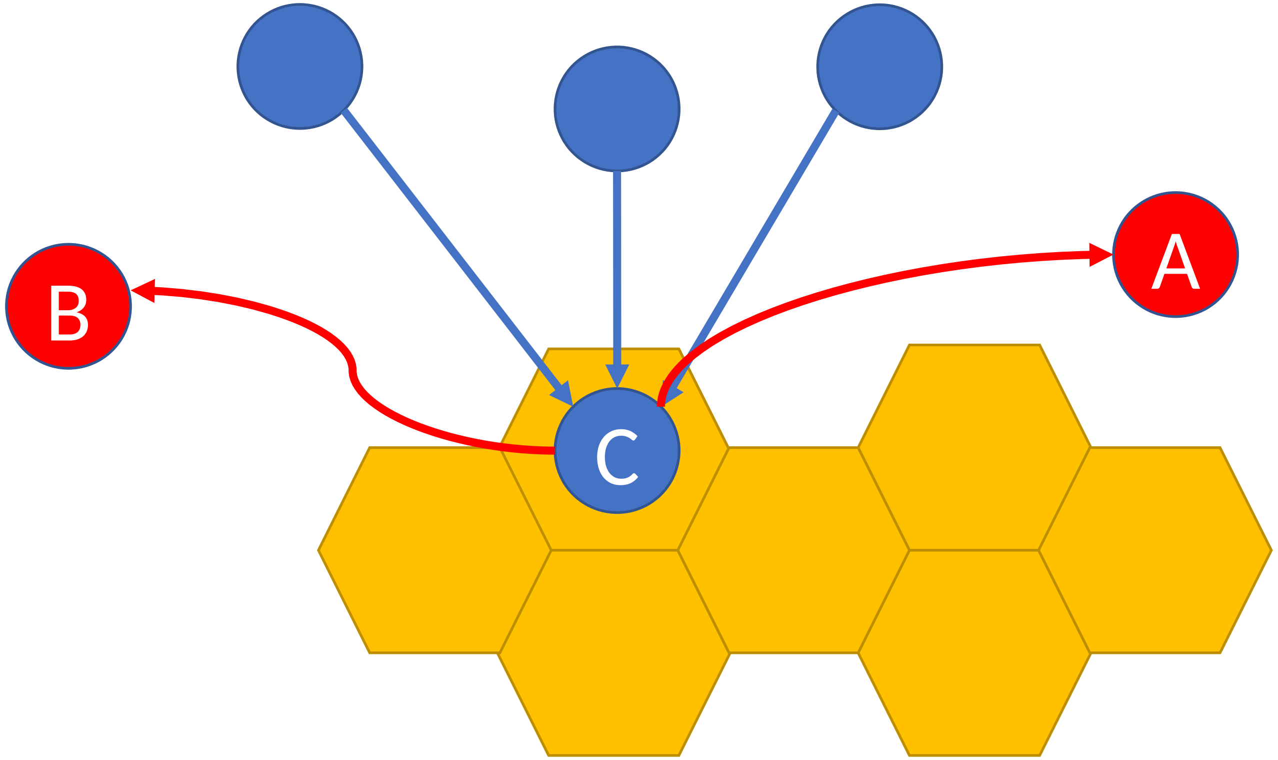
Task	State
A	Running

Hive recognizes speculatively moot tasks



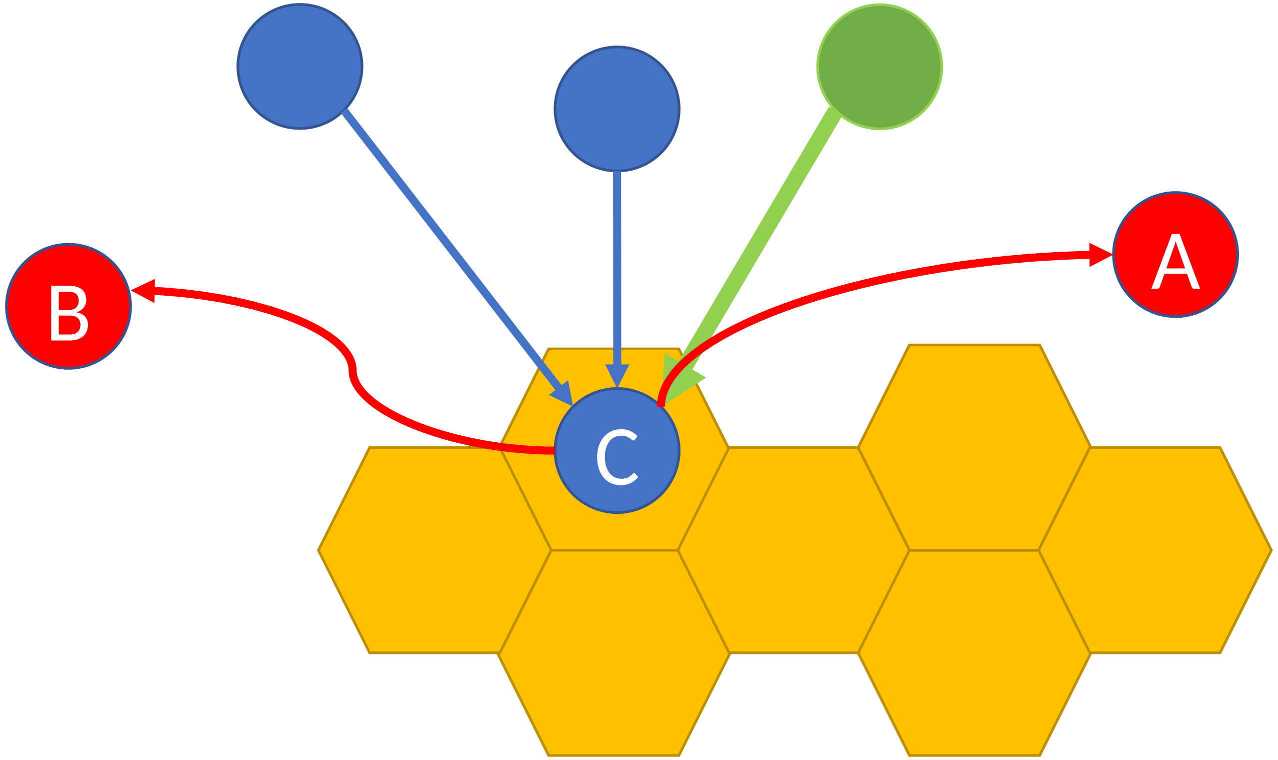
Task	State
A	Moot
B	Running

Hive recognizes speculatively moot tasks



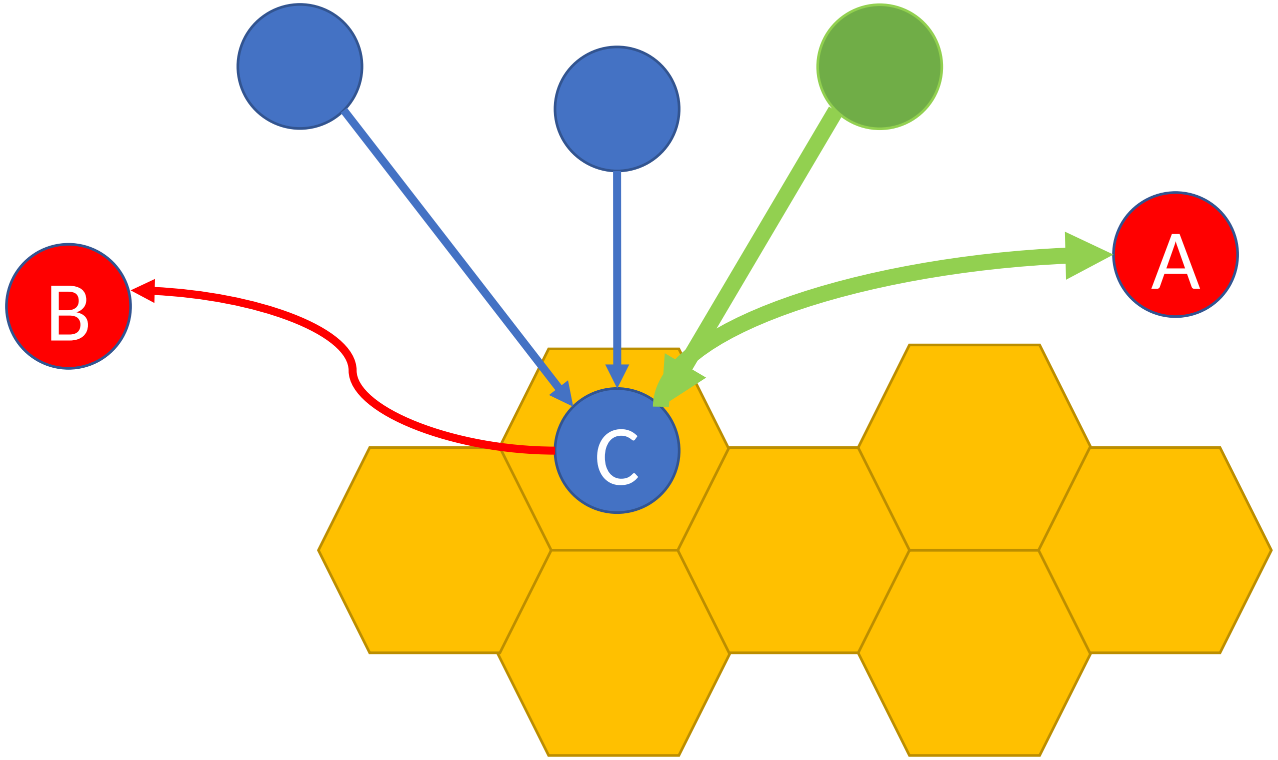
Task	State
A	Moot
B	Moot
C	Running

Hive discards moot tasks as early as possible



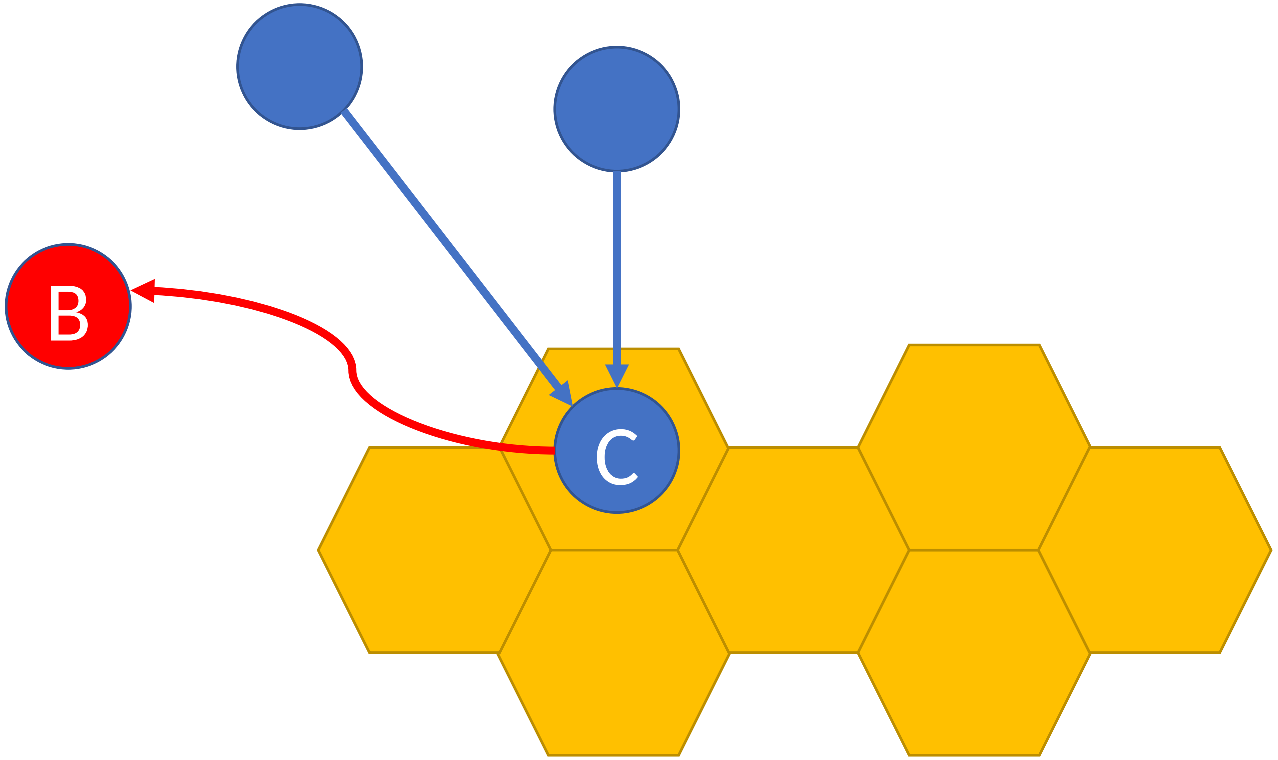
Task	State
A	Moot
B	Moot
C	Running

Hive discards moot tasks as early as possible



Task	State
A	Moot
B	Moot
C	Running

Hive discards moot tasks as early as possible

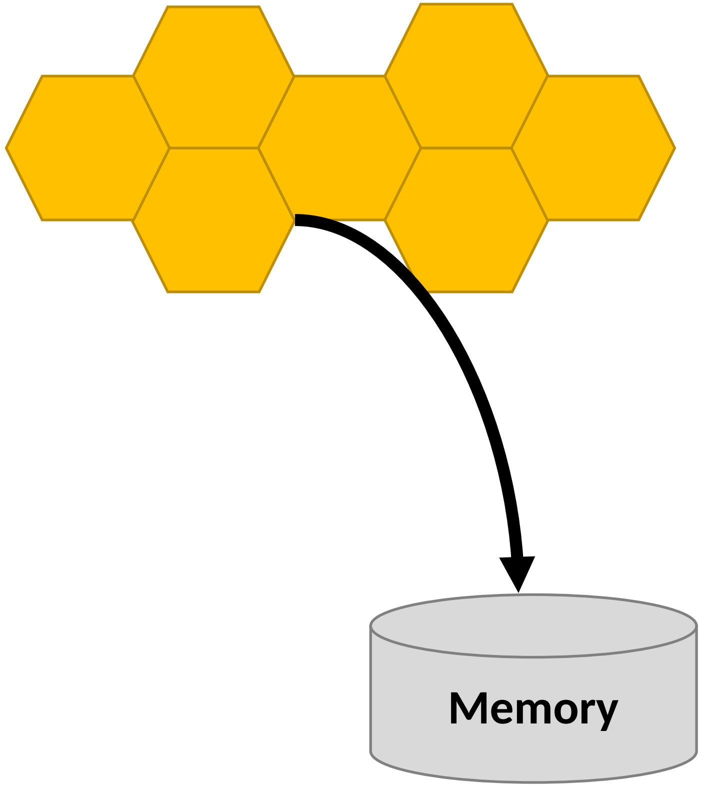


Task	State
B	Moot
C	Running

Moot Tasks Can Be Dropped Without Ever Attempting To Run

Hive adds little area over Swarm

		Task Queue	Task Send Buffer	Commit Queue filters (2–port)	Commit Queue other	Order Queue (TCAM)	Object Map (CAM)
Entries		256	96	64	64	256	256
Entry Size (bytes)	S	51	45	16×32	36	2×8	N/A
	H	65	67	16×32	36	2×8	8
Size (KB)	S	12.75	4.22	32	2.25	4	0
	H	16.25	6.28	32	2.25	4	2
Est. area (mm^2)	S	0.032	0.016	0.149	0.009	0.175	0
	H	0.043	0.028	0.149	0.009	0.175	0.011



The scheduler dependence is distinct

Useful Task

Moot Task

Data

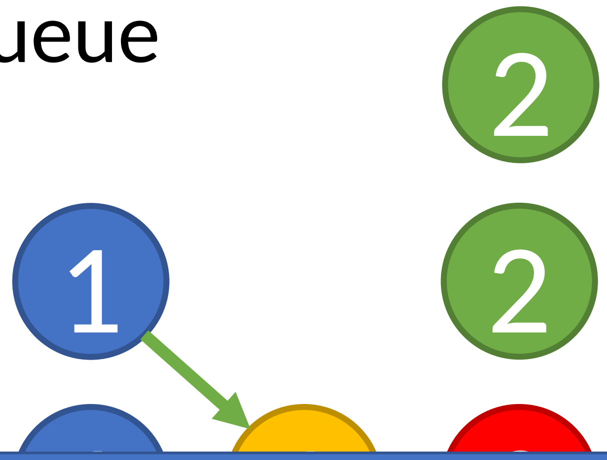
Control

Scheduler

```
while (!pq.empty()) {
  int v, int prio = pq.dequeueM[redacted]
  if (prios[v] < prio) continue[redacted]
  coreness[v] = prio;
  for (int nbr : G.edges[v])
    if (prios[nbr] > prio) {
      prios[nbr]--;
      pq.enqueue(nbr, prios[nbr])
}}
```

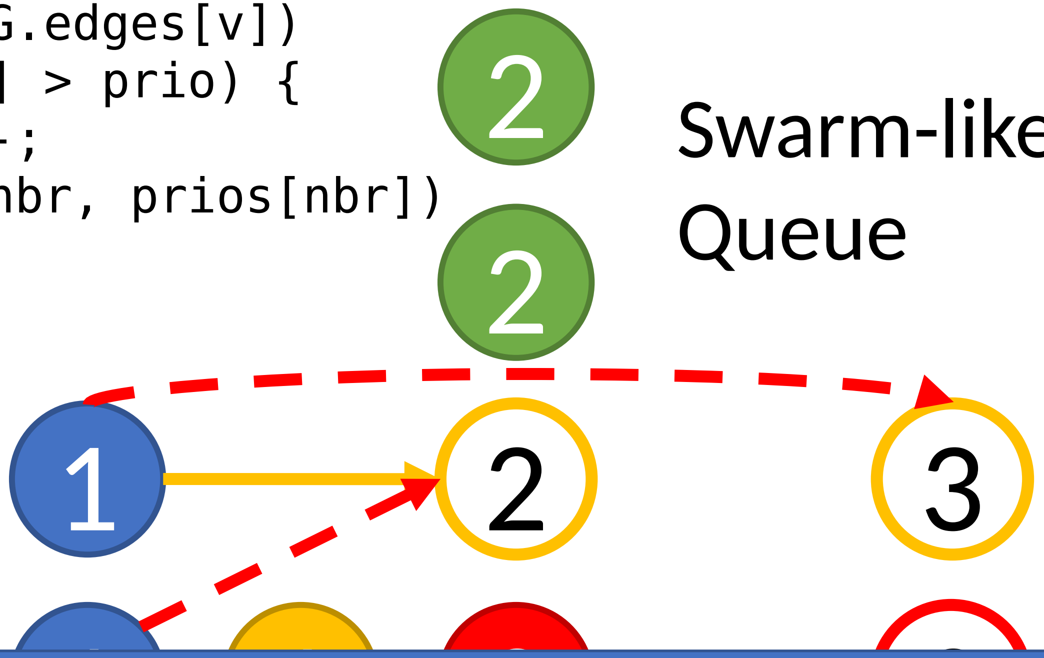
```
CMP R1, #0
ADDEQ R2,R2✉R2
ADDNE R2,R3✉R2
```

Updateable Queue



Priority

Swarm-like Queue

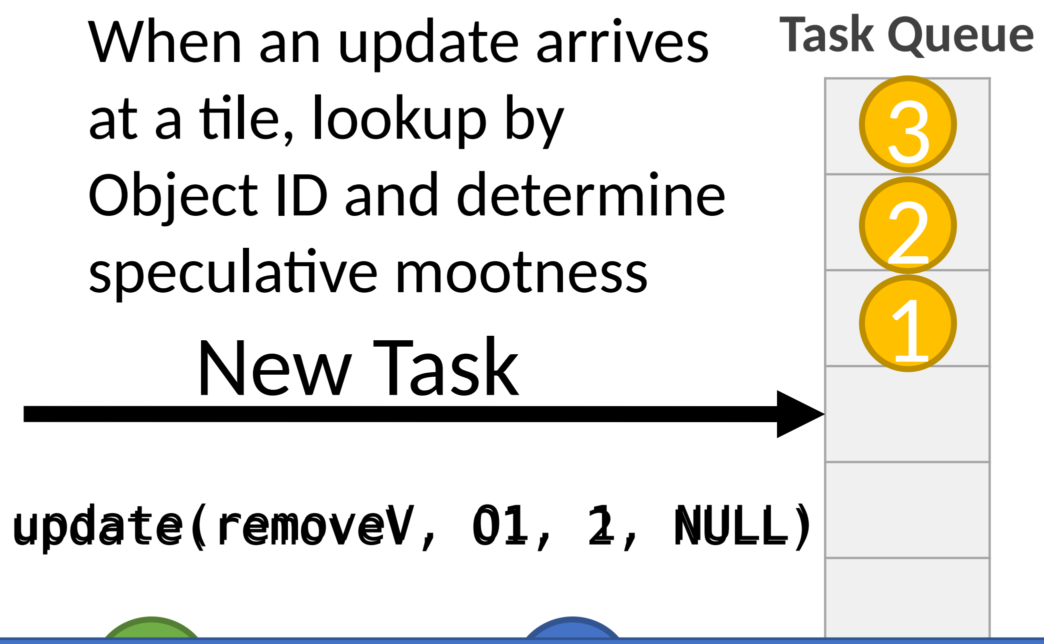


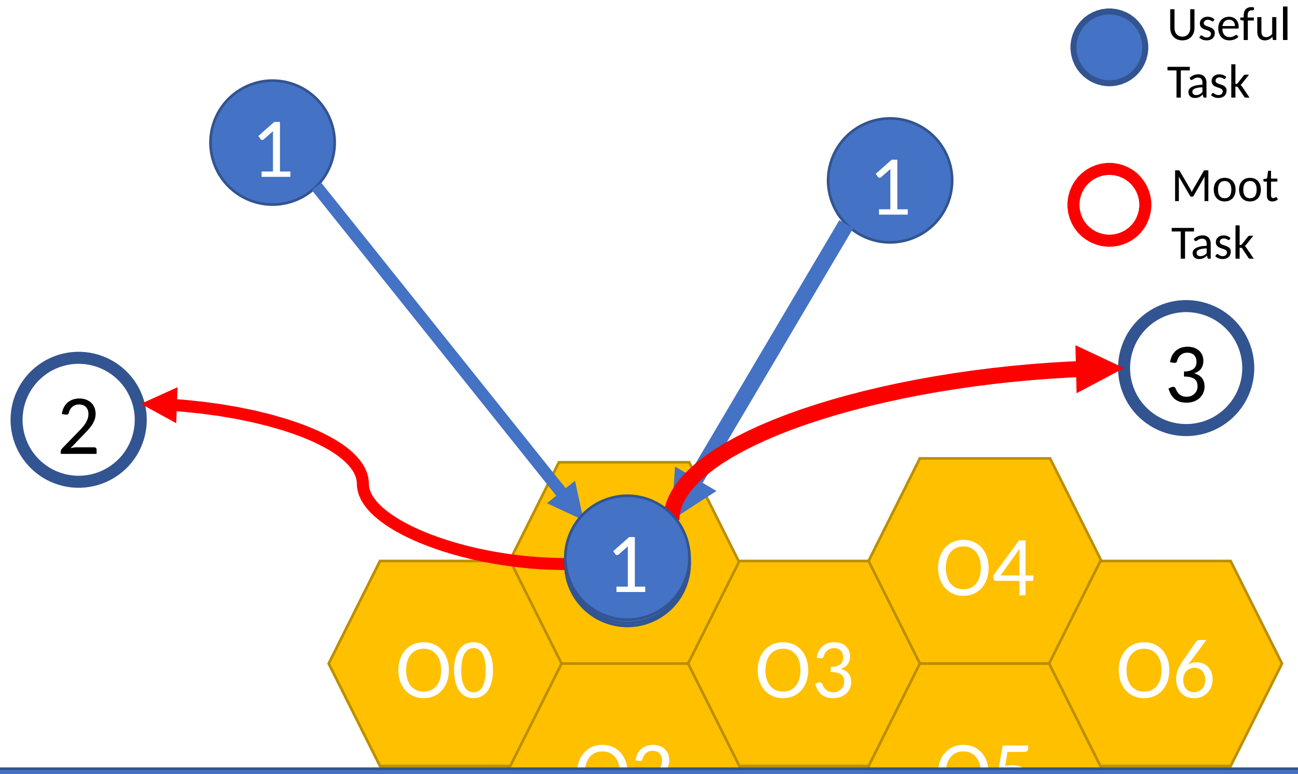
Priority

Predication converts scheduler dependences to data/control

Task versioning enables schedule speculation

When an update arrives at a tile, lookup by Object ID and determine speculative mootness





Task versioning requires a single set of timestamp comparisons on arrival

Moot tasks are discarded before they would have committed if they ran

Converting sequential code to Hive is simple

```
PriorityQueue pq;
for (int v: G.V)
  pq.enqueue(v, G.degree[v]);
while (!pq.empty()) {
  int v, int prio = pq.dequeueMin();
  coreness[v] = prio;
  for (int nbr : G.edges[v])
    if (pq.getPrio(nbr) > prio)
      pq.decrementPrio(nbr);
}
```

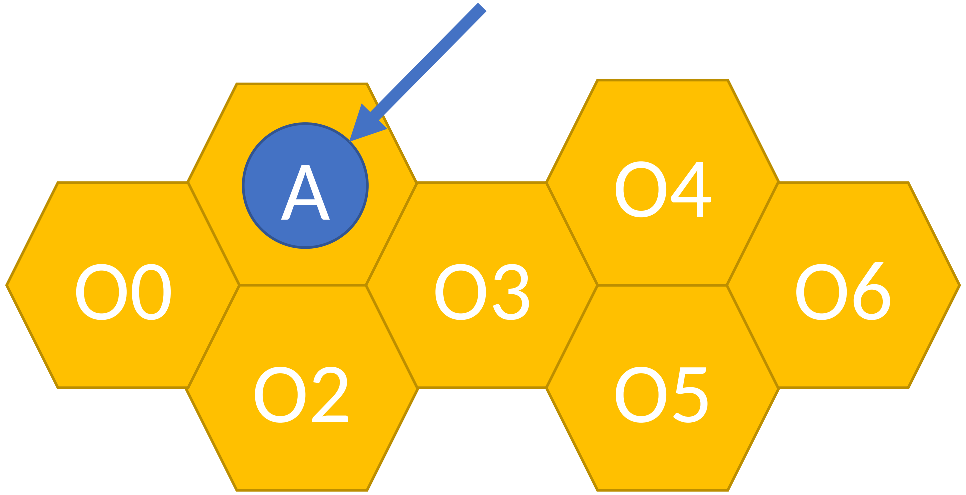
```
void removeV(int v, Timestamp ts) {
  coreness[v] = ts;
  for (int nbr : G.edges[v]) {
    Timestamp prev = hive::getTS(nbr);
    if (prev > ts)
      hive::update(&removeV, nbr, prev-1);
}}

int main() {
  hive::init(G.n)
  for (int v: G.V)
    hive::update(&removeV, v, G.degree[v]);
  hive::run();
}
```

Hive supports priority updates

- Hive programs contain **Tasks** and **Objects**
- Hive **Tasks** contain a function ptr, timestamp, and arguments
  - **Tasks** appear to execute in Timestamp order
- Hive **Objects** are containers for **Tasks**
  - An **Object** holds up to 1 **Task**

```
hive::update(
        fn,   //what to do
        oid,  //what to do it to
        ts,   //when to do it
        args  //what to do it with);
```

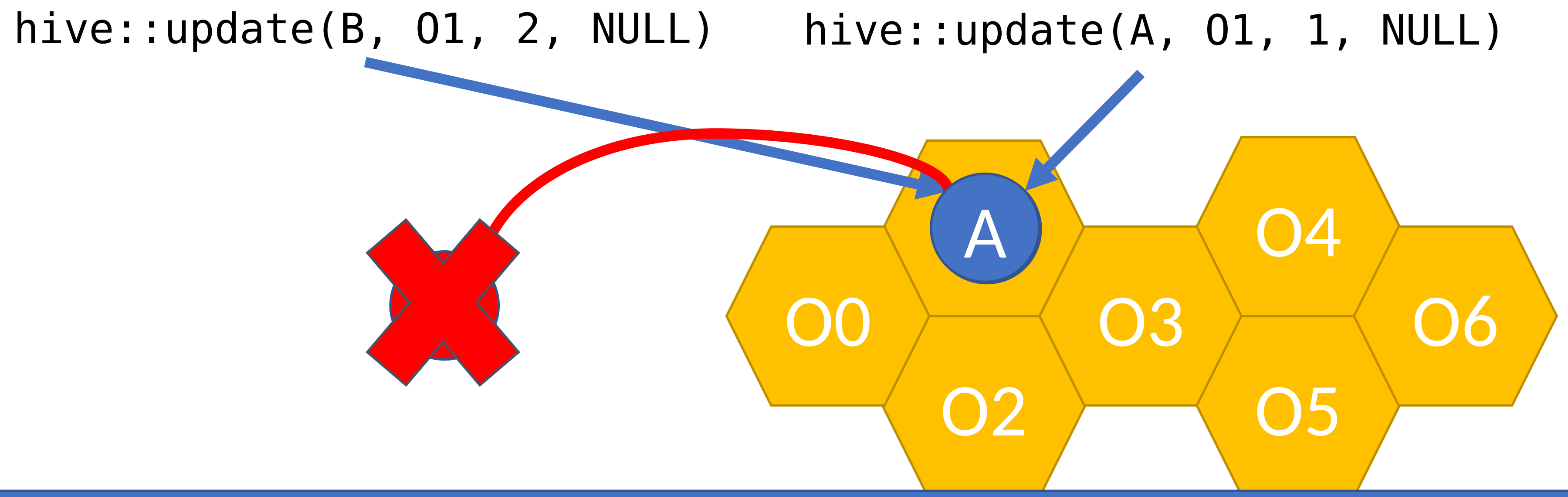
`hive::update(A, O1, 1, NULL)`



Object Table

Hive supports priority updates

When a Hive **Task** updates an **Object** with a **Task** already bound, the old **Task** is discarded and never runs



Hive’s abstract PQ has been updated to contain Task B instead of Task A

Priority queues are hard to parallelize

3 Techniques:

- Bulk-Synchronous [Dhulipala et al. SPAA'17] [Dadu et al. ISCA'21]
  - Very effective when there is a lot of work per barrier
  - Close to sequential when there is little work between barriers
- Relaxed [Aksenov et al. Neurips'20] [Dadu et al. ISCA'21]
  - Can always find parallelism
  - Not always applicable, and loses efficiency as it scales
- Speculation [Blelloch et al. PPoPP'12] [Jeffrey et al. MICRO'15]
  - Always finds parallelism and maintains strict ordering
  - SW speculation has high overheads
  - Existing HW systems do not support priority updates

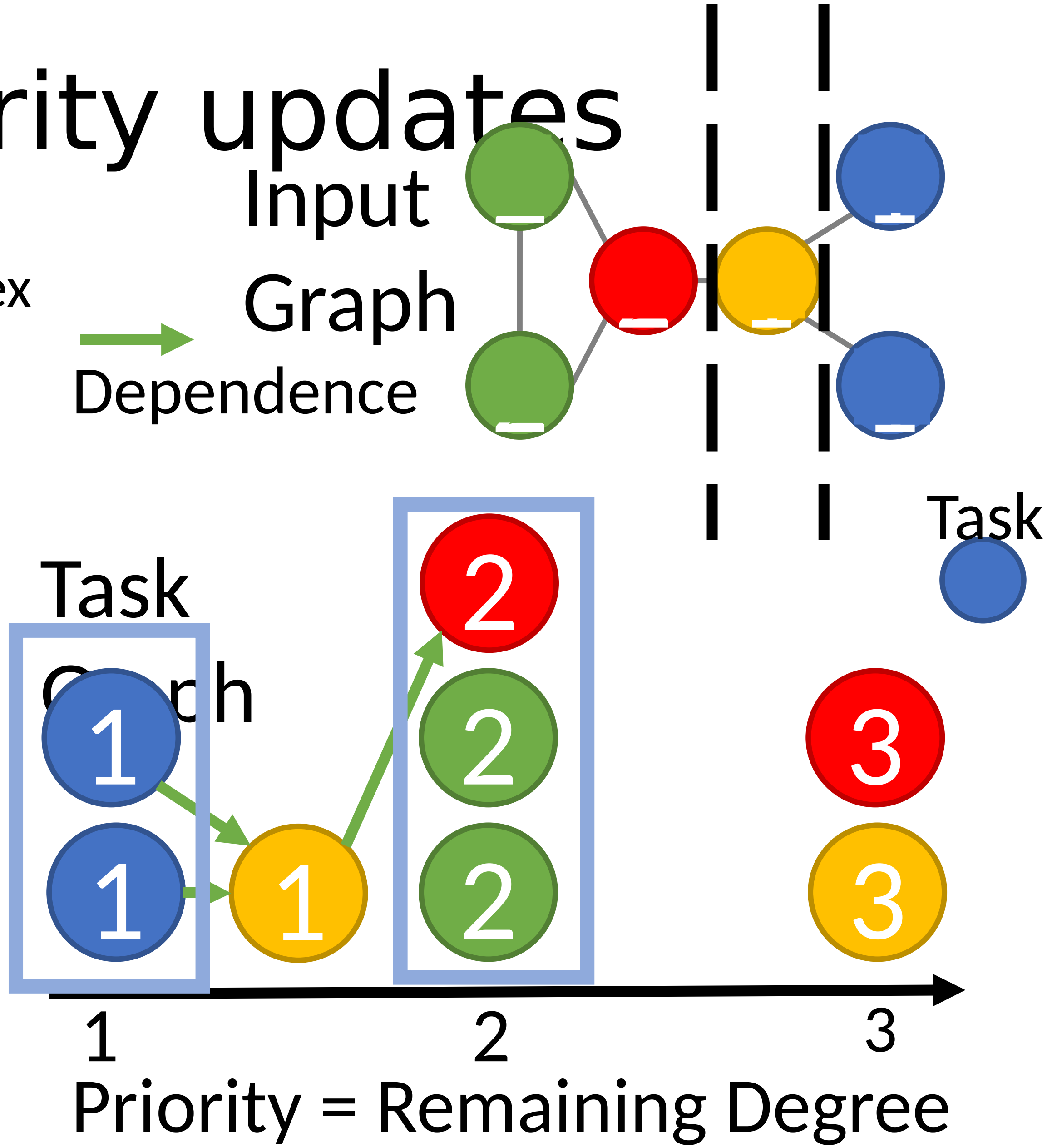
KCore requires priority updates

Maximum core of a vertex ≈ “importance”

To find: repeatedly remove lowest degree vertex

3 Techniques to extract parallelism:

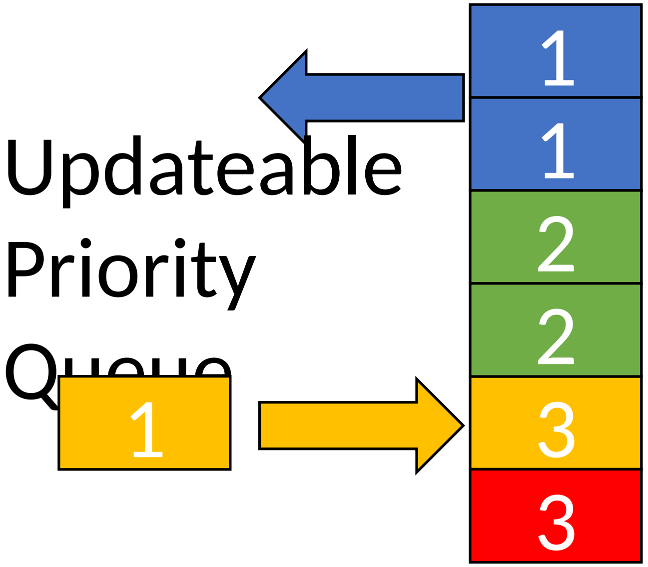
- Bulk-Synchronous [Dhulipala et al. SPAA‘17] [Dadu et al. ISCA’21]
  - Very effective when there is a lot of work per barrier
  - Close to sequential when there is little work between barriers
- Relaxed [Aksenov et al. Neurips’20] [Dadu et al. ISCA’21]
  - Can always find parallelism
  - Not always applicable, and loses efficiency as it scales
- Speculation [Blelloch et al. PPoPP’12] [Jeffrey et al. MICRO’15]
  - Always finds parallelism and maintains strict ordering
  - SW speculation has high overheads
  - Existing HW systems do not support priority updates

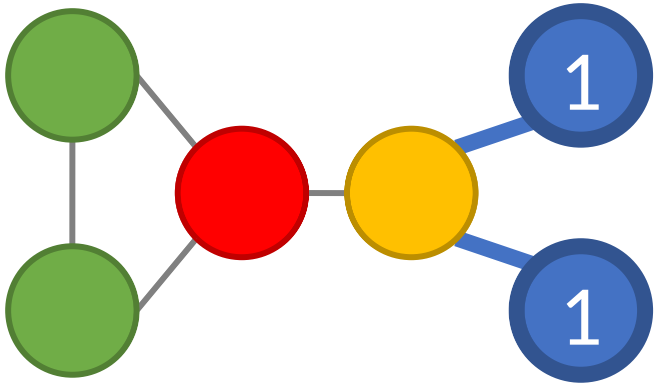


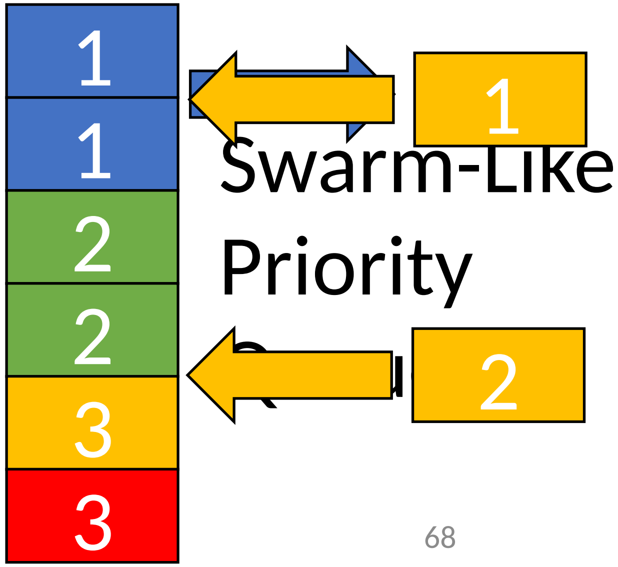
Updateable priority queues are smaller

```
while (!pq.empty()) {
  int v, int prio = pq.dequeueMin();
  coreness[v] = prio;
  for (int nbr : G.edges[v])
    if (pq.getPrio(nbr) > prio)
      pq.decrementPrio(nbr);
}
```

```
while (!pq.empty()) {
  int v, int prio = pq.dequeueMin()
  if (prios[v] < prio) continue;
  coreness[v] = prio;
  for (int nbr : G.edges[v])
    if (prios[nbr] > prio) {
      prios[nbr]--;
      pq.enqueue(nbr, prios[nbr])
}}
```





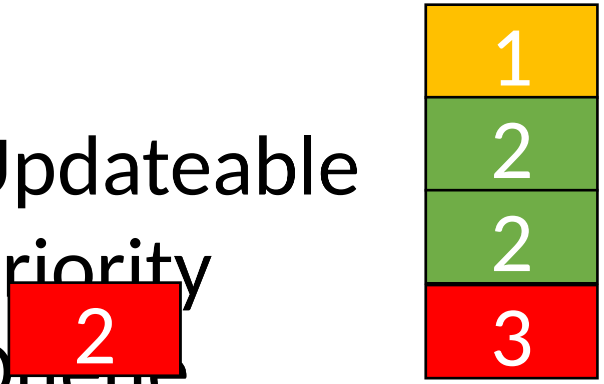


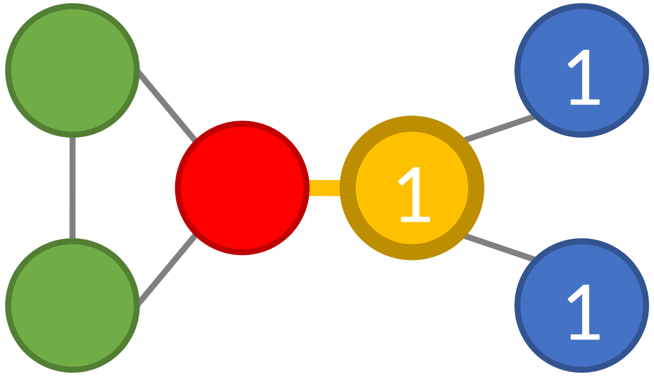
Updateable priority queues are smaller

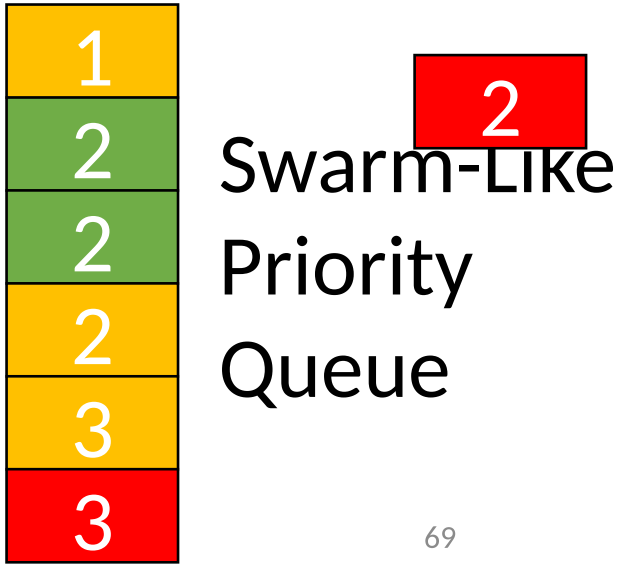
```
while (!pq.empty()) {
  int v, int prio = pq.dequeueMin();
  coreness[v] = prio;
  for (int nbr : G.edges[v])
    if (pq.getPrio(nbr) > prio)
      pq.decrementPrio(nbr);
}
```

```
while (!pq.empty()) {
  int v, int prio = pq.dequeueMin()
  if (prios[v] < prio) continue;
  coreness[v] = prio;
  for (int nbr : G.edges[v])
    if (prios[nbr] > prio) {
      prios[nbr]--;
      pq.enqueue(nbr, prios[nbr])
}}
```

Updateable Priority Queue







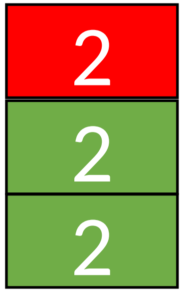
Swarm-Like Priority Queue

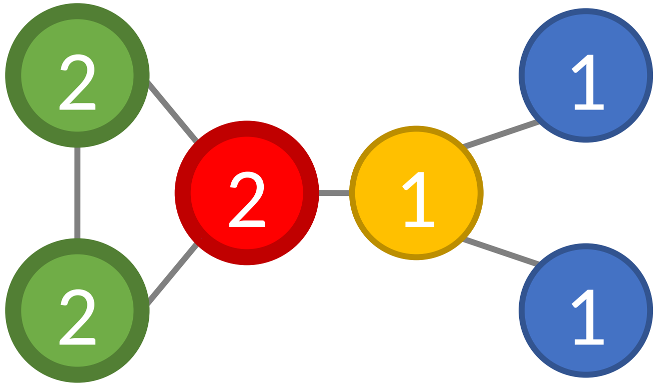
Updateable priority queues are smaller

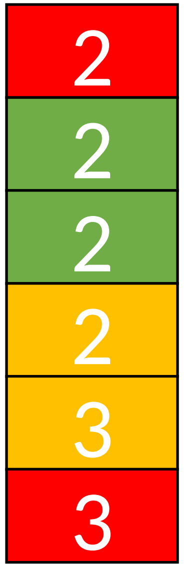
```
while (!pq.empty()) {
  int v, int prio = pq.dequeueMin();
  coreness[v] = prio;
  for (int nbr : G.edges[v])
    if (pq.getPrio(nbr) > prio)
      pq.decrementPrio(nbr);
}
```

```
while (!pq.empty()) {
  int v, int prio = pq.dequeueMin()
  if (prios[v] < prio) continue;
  coreness[v] = prio;
  for (int nbr : G.edges[v])
    if (prios[nbr] > prio) {
      prios[nbr]--;
      pq.enqueue(nbr, prios[nbr])
}}
```

Updateable Priority Queue





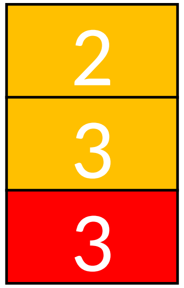


Swarm-Like Priority Queue

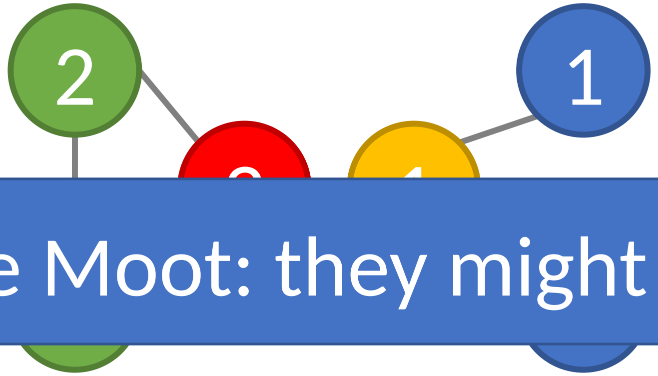
Updateable priority queues are smaller

```
while (!pq.empty()) {
  int v, int prio = pq.dequeueMin();
  coreness[v] = prio;
  for (int nbr : G.edges[v])
    if (pq.getPrio(nbr) > prio)
      pq.decrementPrio(nbr);
}
```

```
while (!pq.empty()) {
  int v, int prio = pq.dequeueMin()
  if (prios[v] < prio) continue;
  coreness[v] = prio;
  for (int nbr : G.edges[v])
    if (prios[nbr] > prio) {
      prios[nbr]--;
      pq.enqueue(nbr, prios[nbr])
}}
```



Updateable Priority



Swarm-Like Priority

Early exiting tasks are Moot: they might as well not run at all

Hive provides ordered tasks with updates

- Hive programs contain **Tasks** and **Objects**
- Hive **Tasks** contain a function ptr, timestamp, and arguments
  - **Tasks** appear to execute in Timestamp order
- Hive **Objects** are containers for **Tasks**
  - An **Object** holds up to 1 **Task**

```
hive::update(&A, O1, 0);
```

```
hive::update(&B, O3, 5, 1, 2);
```

```
hive::update(
      fn,  //what to do
      oid, //what to do it to
      ts,  //when to do it
      args //what to do it with);
```

Object	fn	ts	args
O0	NULL		
O1	A	0	NULL
O2	A	1	NULL
O3	B	5	1,2
O4	NULL		
O5	A	2	NULL
O6	NULL		

Object Table

Hive provides ordered tasks with updates

When a Hive **Task** updates an **Object** with a **Task** already bound, the old **Task** is discarded and never runs

```
hive::update(B, O2, 2, 0, 1)
```

Object	fn	ts	args
O0	NULL		
O1	A	0	NULL
O2	B	2	0,1
O3	B	5	1,2
O4	NULL		
O5	A	2	NULL
O6	NULL		

O2	A		NULL

Hive's schedule can be updated, and replaced tasks do not run

See the paper for…

- A formal scheduler dependence definition
- The relation used to determine Mootness
- Recovery from Mootness misspeculation
- Detecting Mootness in a virtualized queue
- Object table implementation
- And more…